FORMAL REASONING ABOUT SYSTEMS, SOFTWARE AND HARDWARE

Using Functionals, Predicates and Relations

Raymond Boute
INTEC, Ghent University
boute@intec.UGent.be

Abstract Formal reasoning in the sense of "letting the symbols do the work" was Leibniz's dream, but making it possible and convenient for everyday practice irrespective of the availability of automated tools is due to the calculational approach that emerged from Computing Science.

This tutorial provides an initiation in a formal calculational approach that covers not only the discrete world of software and digital hardware, but also the "continuous" world of analog systems and circuits. The formalism (*Funmath*) is free of the defects of traditional notation that hamper formal calculation, yet, by the unified way it captures the conventions from applied mathematics, it is readily adoptable by engineers.

The fundamental part formalizes the equational calculation style found so convenient ever since the first exposure to high school algebra, followed by concepts supporting expression with variables (pointwise) and without (point-free). Calculation rules are derived for (i) proposition calculus, including a few techniques for fast "head" calculation; (ii) sets; (iii) functions, with a basic library of generic functionals that are useful throughout continuous and discrete mathematics; (iv) predicate calculus, making formal calculation with quantifiers as "routine" as with derivatives and integrals in engineering mathematics. Pointwise and point-free forms are covered. Uniform principles for designing convenient operators in diverse areas of discourse are presented. Mathematical induction is formalized in a way that avoids typical errors associated with informal use. Illustrative examples are provided throughout.

The applications part shows how to use the formalism in computing science, including data type definition, systems specification, imperative and functional programming, formal semantics, deriving theories of programming, and also in continuous mathematics relevant to engineering.

Keywords: Analysis, calculational reasoning, data types, functional predicate calculus, Funmath, generic functionals, programming theories, quantifiers

Introduction: motivation and overview

Motivation. Parnas [26] notes that professional engineers can be distinguished from other designers by their ability to use mathematics. In classical (electrical, mechanical) engineering this ability is *de facto* well-integrated. In computing it is still a remote ideal or very fragmented at best; hence the many urgings to integrate formal methods throughout all topics [15, 32]. According to Gopalakrishnan [15], the separate appellation "formal methods" would be redundant if mathematics was practiced in computing as matter-of-factly as in other branches of engineering.

Still, computing needs a more formal mathematical style than classical engineering, as stressed by Lamport [23]. Following Dijkstra [14] and Gries [16], "formal" is taken in the usual mathematical sense of manipulating expressions on the basis of their *form* (syntax) rather than some *interpretation* (semantics). The crucial benefit is the guidance provided by calculation rules, as nicely captured by the maxim *"Ut faciant opus signa"* of the Mathematics of Program Construction conferences [5].

In applied mathematics and engineering, calculation with derivatives and integrals is essentially formal. Readers who enjoyed physics will recall the excitement when calculation pointed the way in case semantic intuition was clueless, showing the value of parallel syntactic intuition. Algebra and analysis tools (Maple, Mathematica etc.) are readily adopted because they stem from formalisms meant for human use (hand calculation), have a unified basis and cover a wide application spectrum.

Comparatively, typical *logical* arguments in theory development are informal, even in computing. Symbolism is often just *syncopation* [29], i.e., using logic symbols as mere shorthands for natural language, such as $\forall$ and $\exists$ abbreviating "for all" and "there exists". This leaves formal logic unexploited as a *reasoning aid* for everyday mathematical practice.

Logic suffers from the historical accident of having had no chance to evolve into a proper calculus for humans [14, 18] before attention shifted to mechanization (even before the computer era). Current logic tools are not readily adopted and need expert users. Arguably this is because they are not based on formalisms suited for human use (which includes "back-of-an-envelope" symbolic calculation). Leading researchers [27] warn that using symbolic tools before mental insight and proficiency in logic is acquired obscures elements that are crucial to understanding.

This tutorial bridges the essential gaps. In particular, it provides a formalism (*Funmath*) by which engineers can calculate with predicates and quantifiers as smoothly as with derivatives and integrals. In addition to direct applicability in everyday mathematical practice whatever the application, it yields superior insight for comparing and using tools.

Overview. Sections 1–3 cover preliminaries and the basis of the formalism: functional predicate calculus and generic functionals. Sections 4–6 show applications in diverse areas of computing and "continuous" mathematics. Due to page limitations, this is more like an extended syllabus, but a full 250-page course text [10] is available from the author.

1. Calculating with expressions and propositions

A *formalism* is a *language* (*notation*) plus formal *calculation rules*. Our formalism needs only four language constructs. Two of these (similar to [17]) are covered here, the other two appear in later sections.

1.1 Expressions, substitution and equality

Syntax conventions. The syntax of *simple expressions* is defined by the following BNF grammar. Underscores designate terminal symbols.

$$\begin{array}{l} \mathit{expression} \;::=\; \mathit{variable} \mid \mathit{constant}_0 \mid \mathit{application} \\ \mathit{application} \;::=\; \underline{(}\mathit{cop}_1\ \mathit{expression}\underline{)} \mid \underline{(}\mathit{expression}\ \mathit{cop}_2\ \mathit{expression}\underline{)} \end{array}$$

Here *variable*, $\mathit{constant}_0$, cop_1 en cop_2 are domain-dependent. Example: with $\mathit{variable} \;::=\; \underline{x} \mid \underline{y} \mid \underline{z}$ and $\mathit{constant}_0 \;::=\; \underline{a} \mid \underline{b} \mid \underline{c}$ and operators defined by $\mathit{cop}_1 \;::=\; \underline{\mathit{succ}} \mid \underline{\mathit{pred}}$ and $\mathit{cop}_2 \;::=\; \underline{+} \mid \underline{\cdot}$, we obtain expressions like 'a' 'x' '$(\mathit{succ}\, y)$' '$(a + y)$' '$((a \cdot (\mathit{succ}\, x)) + y)$'.

When clarity requires, we use quotes ' ' for strings of terminals, and $[\![\ ]\!]$ if metavariables may be present. Lowercase words (e.g., *expression*) designate a nonterminal, the first letter in uppercase (e.g., E) the corresponding syntactic category, i.e., set of symbol strings, and the first letter itself (e.g., e) is a metavariable for a string in that set. Example: let metavariables v, c, ϕ, $\star$ correspond to V, C_0, C_1, C_2, and e, e' to E; then $[\![v]\!]$, $[\![c]\!]$, $[\![(\phi\, e)]\!]$, $[\![(e \star e')]\!]$ represent all forms of simple expressions.

Parentheses can be made optional by the usual conventions. We define *formulas* by $\mathit{formula} \;::=\; \mathit{expression} \equiv \mathit{expression}$, as in $x \cdot y = y \cdot x$.

Substitution. Replacing every occurrence of variable v in expression e by expression d is written $e[v := d]$ and formalized recursively by

Sv:	$w[v := d]$	$= (v = w)\,?\,d \mid w$	for variable v in V
S0:	$c[v := d]$	$= c$	for constant c in C_0
S1:	$(\phi\, e)[v := d]$	$= (\phi\, e[v := d])$	for ϕ in C_1 and e in E
S2:	$(e \star e')[v := d]$	$= (e[v := d] \star e'[v := d])$	for $\star$ in C_2; e, e' in E

All equalities here are purely syntactic (not part of formulas). Expressions like $c\,?\,b \mid a$ (as in **Sv**) are understood as "if c then b, else a". Example: for $(a \cdot \mathit{succ}\, x + y)[x := z \cdot a]$ the rules yield '$a \cdot \mathit{succ}\,(z \cdot a) + y$'.

Multiple (parallel) substitution is a straightforward generalization.

Deduction and equational reasoning Later on we shall see formulas other than equality. Generally, an *inference rule* is a little "table"

$$\frac{Prems}{q}$$

where *Prems* is a set of formulas called *premisses* and q a formula called the *conclusion*. Inference rules are used as follows.

A *consequence* of a set *Hyps* of formulas (called *hypotheses*) is either one of the hypotheses or the conclusion of an inference rule whose premisses are consequences of *Hyps*. A *deduction* is a record of these correspondences. We write $Hyps \vdash q$ if q is a consequence of *Hyps*.

Axioms are selected hypotheses (application-dependent). *Theorems* are consequences of axioms, and *proofs* are deductions of theorems.

The main inference rules are *instantiation* and the *rules for equality*.

a. *Instantiation* (strict): $\dfrac{p}{p[v := e]}$

b. *Leibniz's principle* (not strict): $\dfrac{d = d'}{e[v := d'] = e[v := d'']}$

c. *Symmetry* of equality (not strict): $\dfrac{e'' = e'}{e' = e''}$

d. *Transitivity* of equality (not strict): $\dfrac{e = e', e' = e''}{e = e''}$

A *strict* inference rule requires that its premisses are *theorems*.

In the *equational style*, deductions are recorded in the format

$$\begin{array}{lll} e & =\langle \text{justification} \rangle & e' \\ & =\langle \text{justification} \rangle & e'' \qquad \text{etc.} \end{array} \tag{1}$$

The inference rules are fitted into this format as follows.

a. *Instantiation* In equational reasoning, premiss p is a theorem of the form $d' = d''$, hence the conclusion $p[v := e]$ is $(d' = d'')[v := e]$ which has the form $e' = e''$. Example: $(a + b) \cdot c \;\; =\langle x \cdot y = y \cdot x \rangle \;\; c \cdot (a + b)$.

b. *Leibniz* Premiss p is of the form $d' = d''$ and the conclusion is $e[v := d'] = e[v := d'']$, which has the form $e' = e''$. Example: with premiss $y = a \cdot x$ we may write $x + y \;\; =\langle y = a \cdot x \rangle \;\; x + a \cdot x$.

c. *Symmetry* Premiss p is of the form $e'' = e'$ and the conclusion is $e' = e''$. However, this simple step is usually taken tacitly.

d. *Transitivity* has two equalities for premisses. It is used implicitly to justify chaining $e = e'$ and $e' = e''$ as in (1) to conclude $e = e''$.

1.2 Pointwise and point-free styles of expression

One can specify functions *pointwise* by referring to points in the domain, as in $square\, x = x^2$, or *point-free* using functionals, as in $square = multiply \circ duplicate$ (comment needed nor given at this stage).

The respective archetypes of these styles are *lambda terms* and *combinator terms*, briefly discussed next to capture the essence of symbolic manipulation in both styles in an application-independent form.

Syntax of lambda terms. Bound and free occurrences. The syntax for *lambda terms* [2] is defined by the following BNF grammar.

$$\mathit{term} \quad ::= \quad \mathit{variable} \mid \underline{(}\mathit{term}\ \mathit{term}\underline{)} \mid \underline{(}\underline{\lambda}\mathit{variable}\underline{.}\mathit{term}\underline{)}. \tag{2}$$

Examples: x (xy) $(\lambda x.(((y(\lambda x.(\lambda y.((xz)(\lambda z.((xy)z))))))y)z))$

Naming convention Λ is the syntactic category and $L..R$ metavariables for terms; u, v, w metavariables for variables; x, y, z are typical variables, and symbols like **C**, **D**, **I**, **K**, **S** abbreviate often-used terms.

Terminology A term like (MN) is an *application*, $(\lambda v.M)$ is an *abstraction*: $\lambda v.$ is the *abstractor* and M (the *scope* of $\lambda v.$) the *abstrahend*.

Parentheses convention Outer parentheses are optional in (MN) and in $(\lambda v.M)$ if these terms stand alone or as an abstrahend. Hence the scope extends as far as parentheses permit. Application associates to the left, (LMN) standing for $((LM)N)$. Nested abstractions like $\lambda u.\lambda v.M$ are written $\lambda uv.M$. Example: $\lambda x.y(\lambda xy.xz(\lambda z.xyz))yz$ stands for $(\lambda x.(((y(\lambda x.(\lambda y.((xz)(\lambda z.((xy)z))))))y)z))$, saving 18 parentheses.

Bound and free occurrences Every occurrence of v in $\lambda v.M$ is *bound*. Occurrences that are not bound are *free*. Example: numbering variable occurrences in $\lambda x.y(\lambda xy.xz(\lambda z.xyz))yz$ from 0 tot 11, the only free ones are those of y and z at places 1, 5, 10 and 11. We write $\varphi\, M$ for the set of variables with free occurrences in M, for instance $\varphi\,\text{`}\lambda z.xyz\text{'} = \{\underline{x}, \underline{y}\}$.

Substitution and calculation rules (lambda-conversion). Substituting L for w in M, written $M[w := L]$ or $M[^w_L$, is defined recursively:

$$\begin{array}{rrcl}
\textbf{Svar:} & v[^w_L & = & (v = w)\,?\,L \mid [\![v]\!] \\
\textbf{Sapp:} & (MN)[^w_L & = & (M[^w_L\, N[^w_L) \\
\textbf{Sabs:} & (\lambda v.M)[^w_L & = & (\lambda u.M[^v_u[^w_L) \quad \text{for fresh } u
\end{array}$$

The fresh variable u in **Sabs** prevents free variables in L becoming bound by $\lambda v.$, as in the *erroneous* elaboration $(\lambda x.xy)[^y_x = \lambda x.xx$, which should have been $(\lambda x.xy)[^y_x = \lambda z.zx$.

The *calculation rules* firstly are those for equality: *symmetry*, *transitivity* and *Leibniz's principle*, i.e., $\frac{M=N}{L[v:=M]=L[v:=N]}$. Proper axioms are:

$$\begin{array}{ll}
\alpha\textbf{-conversion:} & (\lambda v.M) = (\lambda w.M[^v_w) \quad \text{provided } w \notin \varphi\, M \\
\beta\textbf{-conversion:} & (\lambda v.M)N = M[^v_N
\end{array}$$

For instance, $(\lambda xy.xy) =_\alpha \lambda xz.xz$ and $(\lambda xy.xy)y =_\beta \lambda z.yz$.

Additional axioms yield variants. Examples are: *rule* ξ: $\frac{M=N}{\lambda v.M=\lambda v.N}$, *rule* η (or η**-conversion**): $(\lambda v.Mv) = M$ (provided $v \notin \varphi M$) and *rule* ζ (*extensionality*): $\frac{Mv=Nv}{M=N}$ provided $v \notin \varphi(M,N)$. As an additional axiom (assuming α and β), rule ζ is equivalent to ξ and η combined.

Henceforth we assume α, β and extensionality, i.e., "everything". Examples of η-conversion are $(\lambda x.yzx) =_\eta yz$ and $\lambda xy.xy =_\eta \lambda x.x$.

Redexes, normal forms and closed terms. A term like $(\lambda v.M)N$ is a *β-redex* and $\lambda v.Mv$ (with $v \notin \varphi M$) is a *η-redex*. A *$\beta\eta$-normal form* (or just "normal form") is a term not containing a β- or η-redex. A term "has a normal form" if it can be reduced to a normal form. According to the *Church-Rosser theorem*, a term has at most one normal form. The term $(\lambda x.xx)(\lambda x.xx)$ even has none.

Closed terms or (lambda-)*combinators* are terms without free variables. Beta-conversion can be encapsulated by properties expressed using metavariables. For instance S, standing for $\lambda xyz.xz(yz)$, has property $\mathsf{S}PQR = PR(QR)$ by β-conversion.

Expressions without variables: combinator terms. Syntax:

$$term \ ::= \ \underline{\boldsymbol{K}} \mid \underline{\boldsymbol{S}} \mid \underline{(}term\, term\underline{)} \tag{3}$$

where $\boldsymbol{K}$ and $\boldsymbol{S}$ are *constants* (using different font to avoid confusion with lambda-combinators). As before, LMN stands for $((LM)N)$.

The calculation rules firstly are those for equality. By lack of variables, Leibniz's principle is $\frac{M=N}{LM=LN}$ and $\frac{M=N}{ML=NL}$. The proper axioms are

$$\boldsymbol{K}LM = L \quad \text{and} \quad \boldsymbol{S}PQR = PR(QR)$$

and *extensionality*: if M en N satisfy $ML = NL$ for any L, then $M = N$.

E.g., $\boldsymbol{SK}MN \ = \langle \boldsymbol{S}PQR = PR(QR)\rangle \ \boldsymbol{K}N(MN) \ = \langle \boldsymbol{K}LM = L\rangle \ N$. Hence, defining $\boldsymbol{I}$ as $\boldsymbol{SKK}$ yields an identity operator: $\boldsymbol{I}N = N$.

Converting combinator terms into (extensionally) equal lambda combinators is trivial. For the reverse, define for every w an operator $[w]$:

For variables w:	$[w]w = \boldsymbol{I}$	(Rule I)
For variables v $(\neq w)$:	$[w]v = \boldsymbol{K}v$	(Rule K')
For constants c:	$[w]c = \boldsymbol{K}c$	(Rule K")
For applications:	$[w](MN) = \boldsymbol{S}([w]M)([w]N)$	(Rule S)
For abstractions:	$[w](\lambda v.M) = [w]([v]M)$	(Rule *abs*)

The crucial property of this operator is $\lambda w.M = [w]M$.

There are two important shortcuts: provided $w \notin \varphi M$, we can use $[w](Mw) = M$ (Rule η) and $[w]M = \boldsymbol{K}M$ (Rule K), the latter being a more efficient replacement for both (K') and (K"). Example: $\lambda xyz.xzy$ $=\langle \lambda w.M = [w]M\rangle \ [x]([y]([z]xzy)) \ =\langle \text{rules}\rangle \ \boldsymbol{S}(\boldsymbol{S}(\boldsymbol{KS})(\boldsymbol{S}(\boldsymbol{KK})\boldsymbol{S}))(\boldsymbol{KK})$.

1.3 Calculational proposition logic

The syntax is that of simple expressions, now with propositional operators. The generic inference rule is *instantiation*. Equality is postponed. We introduce the propositional operators one by one, each with its corresponding axioms and (for $\Rightarrow$ only) its inference rule.

0. Implication ($\underline{\Rightarrow}$). Inference rule: *Modus Ponens:* $\frac{p\,,\,p \Rightarrow q}{q}$ (MP).

Axioms: *Weakening:* $x \Rightarrow y \Rightarrow x$
Distributivity: $(x \Rightarrow y \Rightarrow z) \Rightarrow (x \Rightarrow y) \Rightarrow (x \Rightarrow z)$

Convention: $p \Rightarrow q \Rightarrow r$ stands for $p \Rightarrow (q \Rightarrow r)$, *not* for $(p \Rightarrow q) \Rightarrow r$. Each stage yields a collection of properties (theorems), e.g., at stage **0**:

Idempotency of $\Rightarrow$:	$x \Rightarrow x$
Right isotony of $\Rightarrow$:	$(x \Rightarrow y) \Rightarrow (z \Rightarrow x) \Rightarrow (z \Rightarrow y)$
MP as a formula:	$x \Rightarrow (x \Rightarrow y) \Rightarrow y$
Shunting with $\Rightarrow$:	$(x \Rightarrow y \Rightarrow z) \Rightarrow y \Rightarrow x \Rightarrow z$
Left antitony of $\Rightarrow$:	$(x \Rightarrow y) \Rightarrow (y \Rightarrow z) \Rightarrow (x \Rightarrow z)$
Absorption for $\Rightarrow$:	$(x \Rightarrow x \Rightarrow y) \Rightarrow x \Rightarrow y$

Naming properties is very convenient for invoking them as calculation rules. The properties allow chaining calculation steps by $\Rightarrow$ as in (1).

Very convenient is the *deduction theorem*: if $p \vdash q$ then $\vdash (p \Rightarrow q)$. It allows proving $p \Rightarrow q$ by assuming p as hypothesis (even if p is not a theorem, but then it may not be instantiated) and deducing q.

Henceforth Leibniz's principle will be written $d' = d'' \Rightarrow e[^{v}_{d'} = e[^{v}_{d''}$.

1. Negation ($\underline{\neg}$). Axiom: *Contrapositivity:* $(\neg\, x \Rightarrow \neg\, y) \Rightarrow y \Rightarrow x$. We write $\neg^n$ for n-fold negation: $\neg^0\, p = p$ and $\neg^{n+1}\, p = \neg\, (\neg^n\, p)$.

This stage yields the following main properties.

Contradictory antecedents:	$\neg\, x \Rightarrow x \Rightarrow y$
Skew idempotency of $\Rightarrow$:	$(\neg\, x \Rightarrow x) \Rightarrow x$
Double negation:	$\neg^2\, x \Rightarrow x$ and $x \Rightarrow \neg^2\, x$
Reverse contrapositive:	$(x \Rightarrow y) \Rightarrow \neg\, y \Rightarrow \neg\, x$
Strengthened contrapositive:	$(\neg\, x \Rightarrow \neg\, y) \Rightarrow (\neg\, x \Rightarrow y) \Rightarrow x$
Dilemma:	$(x \Rightarrow y) \Rightarrow (\neg\, x \Rightarrow y) \Rightarrow y$

Note: $\Rightarrow$ and $\neg$ form a complete logic; all further stages are just luxury.

2. Truth constant ($\underline{1}$) with axiom: 1; **falsehood constant ($\underline{0}$)** with axiom: $\neg\, 0$. Typical properties:

Left identity and *right zero* of $\Rightarrow$:	$(1 \Rightarrow x) \Rightarrow x$ and $(x \Rightarrow 1) \Rightarrow 1$
Corresponding laws for constant 0:	$(0 \Rightarrow x) \Rightarrow \neg\, x$ and $(0 \Rightarrow x) \Rightarrow 1$

The rules thus far are sufficient for proving the following

$$\begin{array}{lll} \textsc{Lemma}: & \text{(a)} \quad \neg\, x \Rightarrow p[^x_0 \Rightarrow p & \text{and} \quad \neg\, x \Rightarrow p \Rightarrow p[^x_0; \\ & \text{(b)} \quad x \Rightarrow p[^x_1 \Rightarrow p & \text{and} \quad x \Rightarrow p \Rightarrow p[^x_1 \end{array}$$

The proof uses induction on the structure of p (a variable, a constant, an implication $q \Rightarrow r$, or a negation $\neg\, q$). An immediate consequence is

$$\textsc{Theorem}, \text{ Case Analysis:} \quad p[^x_0\, , p[^x_1 \vdash p.$$

This is the "battering ram" for quickly verifying any conjecture or proving any further theorem in propositional calculus, often by inspection.

3. Logical equivalence (equality) ($\equiv$). The axioms are:

$$\begin{array}{ll} \textit{Antisymmetry of} \Rightarrow: & (x \Rightarrow y) \Rightarrow (y \Rightarrow x) \Rightarrow (x \equiv y) \\ \textit{Weakening of} \equiv: & (x \equiv y) \Rightarrow x \Rightarrow y \quad \text{and} \quad (x \equiv y) \Rightarrow y \Rightarrow x \end{array}$$

One can prove that $\equiv$ is reflexive, symmetric, and transitive. Moreover,

$$\textsc{Theorem}, \text{ Leibniz's principle for } \equiv: \quad (x \equiv y) \Rightarrow (p[^v_x \equiv p[^v_y).$$

Hence, formally $\equiv$ is the *equality* operator for propositional expressions. To minimize parentheses, we give $\equiv$ lower precedence than *any* other operator, just as $=$ has lower precedence than *arithmetic* operators.

Theorems for $\Rightarrow$ that have a converse can be reformulated as equalities. A few samples are: shunting $(x \Rightarrow y \Rightarrow z) \equiv (y \Rightarrow x \Rightarrow z)$, contrapositive $(x \Rightarrow y) \equiv (\neg\, y \Rightarrow \neg\, x)$, double negation $\neg^2 x \equiv x$.

Semidistributivity of $\neg$ *over* $\equiv$, namely, $\neg\,(x \equiv y) \equiv (\neg\, x \equiv y)$ and *associativity* of $\equiv$ (not shared with $=$) are other properties.

4. Logical inequality ($\not\equiv$) or, equivalently, **exclusive-OR ($\oplus$).** Axiom: $x \not\equiv y \equiv \neg\,(x \equiv y)$, i.e., the dual of $\equiv$, or $x \oplus y \equiv \neg\,(x \equiv y)$.

This operator is also associative, symmetric, and mutually associative and interchangeable with $\equiv$ as long as the parity of the number of appearances is preserved, e.g., $x \equiv z \not\equiv y \equiv y \not\equiv x \equiv z$.

The final stage introduces the usual logical OR and logical AND.

5. Disjunction ($\vee$). Axiom: $x \vee y \equiv \neg\, x \Rightarrow y$.

Conjunction ($\wedge$). Axiom: $x \wedge y \equiv \neg\,(x \Rightarrow \neg\, y)$.

Main properties are the *rules of De Morgan*: $\neg\,(x \wedge y) \equiv (\neg\, x \vee \neg\, y)$ and $\neg\,(x \vee y) \equiv (\neg\, x \wedge \neg\, y)$, and many rules relating the other operators, including not only the familiar rules of binary algebra or switching algebra, but also often-used rules in calculational logic [13, 17], such as

$$\textit{Shunting with} \wedge: \quad x \Rightarrow y \Rightarrow z \equiv x \wedge y \Rightarrow z$$

1.4 Binary algebra and conditional expressions

The preliminaries conclude with a "concrete" (non-axiomatic) proposition calculus, and calculation rules for conditional expressions.

Binary algebra. Binary algebra views propositional operators ($\Rightarrow$, $\neg$, $\wedge$ etc.) as functions on the set $\mathbb{B}$ of booleans. As explained in [6, 8], we define $\mathbb{B} := \{0, 1\}$ rather than using separate "truth values" like T, F. The main advantage is that this makes binary algebra a subalgebra of *minimax algebra*, namely, the algebra of the *least upper bound* ($\curlyvee$) and *greatest lower bound* ($\curlywedge$) operators over $\mathbb{R}' := \mathbb{R} \cup \{-\infty, +\infty\}$, defining

$$a \curlyvee b \leq c \;\equiv\; a \leq c \wedge b \leq c \quad \text{and} \quad c \leq a \curlywedge b \;\equiv\; c \leq a \wedge c \leq b. \tag{4}$$

A collection of algebraic laws is easily derived by high school algebra. In binary algebra, $\Rightarrow$, $\equiv$, $\vee$, $\wedge$ are restrictions to $\mathbb{B}$ of $\leq$, $=$, $\curlyvee$, $\curlywedge$ [8]. Laws of minimax algebra particularize to laws over $\mathbb{B}$, e.g., from (4):

$$a \vee b \Rightarrow c \;\equiv\; (a \Rightarrow c) \wedge (b \Rightarrow c) \quad \text{and} \quad c \Rightarrow a \wedge b \;\equiv\; (c \Rightarrow a) \wedge (c \Rightarrow b)$$

A variant sharing most (not all) properties is proposed by Hehner [20].

Conditional expressions. This very convenient formulation of conditional expressions is based on the combining the following 3 elements:

(i) Tuples as functions, defining $(a, b, c)\,0 = a$ and $(a, b, c)\,1 = b$ etc.

(ii) Binary algebra embedding propositional calculus in $\mathbb{B}$-arithmetic.

(iii) Generic functionals, in particular *function composition* ($\circ$) defined here by $(f \circ g)\,x = f\,(g\,x)$ and *transposition* ($^{\mathsf{U}}$) with $f^{\mathsf{U}} y\,x = f\,x\,y$. The main properties for the current purpose are the distributivity laws

$$f \circ (x, y, z) = f\,x, f\,y, f\,z \quad \text{and} \quad (f, g, h)^{\mathsf{U}} x = f\,x, g\,x, h\,x.$$

For binary c and any a and b, we now *define* the conditional $c\,?\,b \mid a$ by

$$c\,?\,b \mid a \quad = \quad (a, b)\,c. \tag{5}$$

Simple calculation yields two distributivity laws for conditionals:

$$f\,(c\,?\,x \mid y) = c\,?\,f\,x \mid f\,y \quad \text{and} \quad (c\,?\,f \mid g)\,x = c\,?\,f\,x \mid g\,x. \tag{6}$$

In the particular case where a and b (and, of course, c) are all binary,

$$c\,?\,b \mid a \quad \equiv \quad (c \Rightarrow b) \wedge (\neg\, c \Rightarrow a) \tag{7}$$

Finally, since predicates are functions and $z=$ is a predicate,

$$z = (c\,?\,x \mid y) \quad \equiv \quad (c \Rightarrow z = x) \wedge (\neg\, c \Rightarrow z = y) \tag{8}$$

These laws are all one ever needs for working with conditionals!

2. Introduction to Generic Functionals

2.1 Sets, functions and predicates

Sets and set equality. We treat sets formally, with basic operator $\underline{\in}$ and calculation rules directly defined or derived via proposition calculus, such as $e \in X \cap Y \;\equiv\; e \in X \wedge e \in Y$ and $e \in X \cup Y \;\equiv\; e \in X \vee e \in Y$. The Cartesian product has axiom $(d, e) \in X \times Y \;\equiv\; d \in X \wedge e \in Y$.

Leibniz's principle yields for set elements $d = e \Rightarrow (d \in X \;\equiv\; e \in X)$. In our (higher-order) formalism, we require it for sets as well:

$$\text{Leibniz (sets):}\quad X = Y \;\Rightarrow\; (e \in X \;\equiv\; e \in Y). \tag{9}$$

Equivalently, $(p \Rightarrow X = Y) \Rightarrow p \Rightarrow (e \in X \;\equiv\; e \in Y)$, for proposition p.

The converse is expressed as follows: for fresh variable (tuple) v,

$$\text{Set extensionality:}\quad \frac{p \Rightarrow (v \in X \;\equiv\; v \in Y)}{p \Rightarrow X = Y}. \tag{10}$$

Here p allows embedding extensionality in a calculation chain as

$$\begin{array}{lll} p & \Rightarrow\langle\text{Calculations deriving r.h.s.}\rangle & v \in X \;\equiv\; v \in Y \\ & \Rightarrow\langle\text{Set extensionality}\rangle & X = Y \end{array}$$

cautioning that this should *not* be read as $(v \in X \;\equiv\; v \in Y) \Rightarrow X = Y$.

The *empty* set $\emptyset$ has axiom $x \notin \emptyset$. A singleton set is written $\iota\, e$, with axiom $d \in \iota\, e \;\equiv\; d = e$. We reserve $\{\ \}$ for better purposes discussed later, one consequence being the rule $e \in \{v : X \mid p\} \;\equiv\; e \in X \wedge p[^v_e$.

Functions and predicates. A function f is *not* a set of pairs (which is the *graph* of the function), but a mathematical concept in its own right, fully specified by its *domain* $\mathcal{D}\, f$ and its *mapping*. This is axiomatized by a *domain axiom* and a *mapping axiom*, which are of (or can be rewritten in) the form $x \in \mathcal{D}\, f \;\equiv\; p$ and $x \in \mathcal{D}\, f \Rightarrow q$ respectively. Here q typically is a proposition with f and x, as illustrated in $n \in \mathbb{N} \Rightarrow \mathit{succ}\, n = n + 1$.

In declarative formalisms, types are sets. Notions from programming are too restrictive for mathematics [9, 25]. For instance, if we assume a function *fac* to be specified such that $\mathcal{D}\, \mathit{fac} = \mathbb{N}$, then instantiating

$$n > 0 \quad\Rightarrow\quad \mathit{fac}\, n = n \cdot \mathit{fac}\, (n - 1),$$

with $n := 0$ would be a type error in programming due to the application $\mathit{fac}\, (-1)$, although mathematically this is perfectly sensible.

Since mapping specifications have the form the form $x \in \mathcal{D}\, f \;\Rightarrow\; q$, the consequent q is irrelevant in case $x \notin \mathcal{D}\, f$. Expressions of this form (or $x \in \mathcal{D}\, f \wedge q$ etc.) are called *guarded* [9] and, if properly written, are seen to be "robust" with respect to out-of-domain applications.

A *predicate* P is a $\mathbb{B}$-valued function: $x \in \mathcal{D}\, P \Rightarrow P\, x \in \mathbb{B}$.

Bindings and abstraction. A *binding* has the general form $i : S \mathbin{\mathbb{A}} p$ (the $\mathbin{\mathbb{A}} p$ is optional). It denotes no object by itself, but *introduces* or *declares* a (tuple of) identifiers(s) i, at the same time specifying that $i \in S \wedge p$. For instance, $n : \mathbb{Z} \mathbin{\mathbb{A}} n \geq 0$ is interchangeable with $n : \mathbb{N}$.

As explained elsewhere [10], the common practice of overloading the relational operator $\in$ with the role of binding, as in $\{x \in \mathbb{Z} \mid x < y\}$, can lead to ambiguities, which we avoid by always using : for binding.

Identifiers are *variables* if declared in an *abstraction* (of the form *binding* . *expression*), constants if declared in a *definition* **def** *binding*.

Our abstraction generalizes lambda abstraction by specifying domains:

$$\begin{array}{ll} \text{Domain axiom:} & d \in \mathcal{D}\,(v : X \mathbin{\mathbb{A}} p \,.\, e) \;\equiv\; d \in X \wedge p[^v_d \\ \text{Mapping axiom:} & d \in \mathcal{D}\,(v : X \mathbin{\mathbb{A}} p \,.\, e) \Rightarrow (v : X \mathbin{\mathbb{A}} p \,.\, e)\, d = e[^v_d \end{array} \tag{11}$$

We assume $v \notin \varphi\, X$. Abstraction is also the key to synthesizing familiar expressions such as $\{n : \mathbb{Z} \,.\, 2 \cdot n\}$, $\forall\, x : \mathbb{R} \,.\, x^2 \geq 0$ and $\sum i : 0 \,..\, n \,.\, x^i$.

Function equality. Leibniz's principle in guarded form for domain elements is $d = e \Rightarrow d \in \mathcal{D}\, f \wedge e \in \mathcal{D}\, f \Rightarrow f\, d = f\, e$. For functions:

$$\begin{array}{lrl} \text{Leibniz (functions):} & f = g & \Rightarrow \quad \mathcal{D}\, f = \mathcal{D}\, g \\ & & \wedge \quad (e \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, e = g\, e) \end{array} \tag{12}$$

or $(p \Rightarrow f = g) \Rightarrow p \Rightarrow \mathcal{D}\, f = \mathcal{D}\, g \wedge (e \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, e = g\, e)$. Since this captures all that can be deduced from $f = g$, the converse is:

$$\text{Function extensionality:} \qquad \frac{p \Rightarrow \mathcal{D}\, f = \mathcal{D}\, g \wedge (v \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, v = g\, v)}{p \Rightarrow f = g}. \tag{13}$$

We use (13) in chaining calculation steps as shown for sets.

As an example, let $f := v : X \mathbin{\mathbb{A}} q \,.\, d$ and $g := v : Y \mathbin{\mathbb{A}} r \,.\, e$ (using v in both preserves generality by α-convertibility). Now (11) and (12) yield

$$\begin{array}{l} (v : X \mathbin{\mathbb{A}} q \,.\, d) = (v : Y \mathbin{\mathbb{A}} r \,.\, e) \\ \qquad \Rightarrow \quad (v \in X \wedge q \;\equiv\; v \in Y \wedge r) \wedge (v \in X \cap Y \wedge q \wedge r \Rightarrow d = e) \end{array}$$

Constant functions. Constant functions are trivial but useful. We specify them using the *constant function definer* ($^\bullet$) defined by

$$X^\bullet e \quad = \quad v : X \,.\, e \quad \text{assuming } v \notin \varphi\, e. \tag{14}$$

Equivalently, $x \in \mathcal{D}\,(X^\bullet e) \;\equiv\; x \in X$ and $x \in \mathcal{D}\,(X^\bullet e) \Rightarrow (X^\bullet e)\, x = e$.

Two often-used special forms deserve their own symbol. The *empty function* ε is defined by $\varepsilon = \emptyset^\bullet e$ (regardless of e since $\emptyset^\bullet e = \emptyset^\bullet e'$). The *one-point function definer* ($— \mapsto —$) is defined by $d \mapsto e = \iota\, d^\bullet e$ for any d and e, which is similar to *maplets* in Z [28].

2.2 Concrete generic functionals, first batch

Design principle. Generic functionals [11] support the point-free style but, unlike the untyped combinator terms from section 1.2, take into account function domains. One of them (filtering) is a generalization of $f = x : \mathcal{D}\, f \,.\, f\, x$ (η-conversion) to introduce or eliminate variables; the others can reshape expressions, e.g., to make filtering applicable.

The design principle can be explained by analogy with familiar functionals. For instance, function composition ($\circ$), with $(f \circ g)\, x = f\, (g\, x)$, traditionally requires $\mathcal{R}\, g \subseteq \mathcal{D}\, f$, in which case $\mathcal{D}\, (f \circ g) = \mathcal{D}\, g$. Instead of restricting the argument functions, we define the *domain of the result functions* to contain *exactly those points that do not cause out-of-domain applications* in the image definition. This makes the functionals applicable to all functions in continuous and discrete mathematics.

This first batch contains only functionals whose definition does not require quantification. For conciseness, we use abstraction in the definitions; separation into domain and mapping axioms is a useful exercise.

Function and set filtering ($\downarrow$). For any function f, predicate P,

$$f \downarrow P \quad = \quad x : \mathcal{D}\, f \cap \mathcal{D}\, P \wedge P\, x \,.\, f\, x \tag{15}$$

This captures the usual *function restriction* ($\restriction$): for function f, set X,

$$f \restriction X \quad = \quad f \downarrow (X \bullet 1). \tag{16}$$

Similarly, for any set X we define $x \in (X \downarrow P) \equiv x \in X \cap \mathcal{D}\, P \wedge P\, x$.

We write a_b for $a \downarrow b$. With partial application, this yields a formal basis and calculation rules for convenient shorthands like $f_{<n}$ and $\mathbb{Z}_{>0}$.

Function composition ($\circ$). For any functions f and g,

$$f \circ g \quad = \quad x : \mathcal{D}\, g \wedge g\, x \in \mathcal{D}\, f \,.\, f\, (g\, x). \tag{17}$$

Dispatching (&) [24] and parallel ($\parallel$). For any functions f and g,

$$f \,\&\, g \quad = \quad x : \mathcal{D}\, f \cap \mathcal{D}\, g \,.\, f\, x, g\, x \tag{18}$$

$$f \parallel g \quad = \quad (x, y) : \mathcal{D}\, f \times \mathcal{D}\, g \,.\, f\, x, g\, y \tag{19}$$

(Duplex) direct extension ($\widehat{\ }$). For any functions $\star$ (infix), f, g,

$$f \mathbin{\widehat{\star}} g \quad = \quad x : \mathcal{D}\, f \cap \mathcal{D}\, g \wedge (f\, x, g\, x) \in \mathcal{D}\, (\star) \,.\, f\, x \star g\, x \tag{20}$$

Sometimes we need *half direct extension*: for any function f, any x,

$$f \mathbin{\overleftarrow{\star}} x = f \mathbin{\widehat{\star}} (\mathcal{D}\, f \bullet x) \quad \text{and} \quad x \mathbin{\overrightarrow{\star}} f = (\mathcal{D}\, f \bullet x) \mathbin{\widehat{\star}} f. \tag{21}$$

Simplex direct extension ($\overline{\ }$) is defined by $\overline{f}\, g = f \circ g$.

Function override. For any functions f and g,

$$f \oslash g \;=\; x : \mathcal{D}\, f \cup \mathcal{D}\, g \,.\, x \in \mathcal{D}\, f \,?\, f\, x \upharpoonright g\, x \tag{22}$$

$$f \oslash g \;=\; x : \mathcal{D}\, f \cup \mathcal{D}\, g \,.\, x \in \mathcal{D}\, g \,?\, g\, x \upharpoonright f\, x \tag{23}$$

Function merge ($\uplus$). For any functions f and g,

$$f \uplus g \;=\; x : \mathcal{D}\, f \cup \mathcal{D}\, g \wedge (x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, x = g\, x) \,.\, (f \oslash g)\, x \tag{24}$$

Relational functionals: compatibility ($\copyright$), subfunction ($\subseteq$).

$$\begin{array}{lcl} f \copyright g & \equiv & f \rceil \mathcal{D}\, g = g \rceil \mathcal{D}\, f \\ f \subseteq g & \equiv & f = g \rceil \mathcal{D}\, f. \end{array}$$

Remark on algebraic properties. The operators presented entail a rich collection of algebraic laws that can be expressed in point-free form, yet preserve the intricate domain refinements (as can be verified calculationally). Examples are: for composition, $f \circ (g \circ h) = (f \circ g) \circ h$ and $\overline{h \circ g} = \overline{h} \circ \overline{g}$; for extension, $(\widehat{\star}) = \overline{(\star)} \circ (\&)$. Elaboration is beyond the scope of this tutorial, giving priority to later application examples.

Elastic extensions for generic functionals. *Elastic operators* are functionals that, combined with function abstraction, unobtrusively replace the many ad hoc abstractors from common mathematics, such as $\forall\, x : X$ and $\sum_{i=m}^{n}$ and $\lim_{x \to a}$. If an elastic operator F and (infix) operator $\star$ satisfy $F\,(x, y) = x \star y$, then F is an *elastic extension* of $\star$. Such extensions are not unique, leaving room for judicious design, as illustrated here for some two-argument generic functionals.

Transposition. Noting that $(g \,\&\, h)\, x\, i = (g, h)\, i\, x$ for i in $\mathbb{B}$ suggests taking *transposition* $({}^T)$ for the elastic extension of &, in view of the argument swap in $f^T y\, x = f\, x\, y$. Making this generic requires deciding on the definition of $\mathcal{D}\, f^T$ for *any* function family f. For & we want $\mathcal{D}\, f^{\mathsf{T}} = \bigcap x : \mathcal{D}\, f \,.\, \mathcal{D}\,(f\, x)$ or, in point-free style, $\mathcal{D}\, f^{\mathsf{T}} = \bigcap (\mathcal{D} \circ f)$. For the most "liberal" design, union is the choice. Elaborating both yields

$$f^{\mathsf{T}} \;=\; y : \bigcap (\mathcal{D} \circ f) \,.\, x : \mathcal{D}\, f \,.\, f\, x\, y \tag{25}$$

$$f^{\mathsf{U}} \;=\; y : \bigcup (\mathcal{D} \circ f) \,.\, x : \mathcal{D}\, f \wedge (y \in \mathcal{D}\,(f\, x)) \,.\, f\, x\, y \tag{26}$$

Parallel ($\|$). For any function family F and function f,

$$\| F\, f \;=\; x : \mathcal{D}\, F \cap \mathcal{D}\, f \wedge f\, x \in \mathcal{D}\,(F\, x) \,.\, F\, x\,(f\, x) \tag{27}$$

This is a typed variant of the $\boldsymbol{S}$-combinator from section 1.2.

3. Functional Predicate Calculus

3.1 Axioms and basic calculation rules

Axioms. A *predicate* is a $\mathbb{B}$-valued function. We define the *quantifiers* $\forall$ and $\exists$ as predicates over predicates. For any predicate P:

$$\forall\, P \;\equiv\; P = \mathcal{D}\, P^\bullet 1 \quad \text{and} \quad \exists\, P \;\equiv\; P \neq \mathcal{D}\, P^\bullet 0 \tag{28}$$

The point-free style is chosen for clarity. The familiar forms $\forall\, x : X \,.\, p$ is obtained by taking for P a predicate $x : X \,.\, p$ where p is a proposition.

Most derived laws are equational. The proofs for the first few laws require separating $\equiv$ into $\Rightarrow$ and $\Leftarrow$, but the need to do so will diminish as laws accumulate, and vanishes by the time we reach applications.

Calculation example. Function equality (12, 13) as one equation.

$$\text{Theorem, Function equality:}\quad f = g \;\equiv\; \mathcal{D}\, f = \mathcal{D}\, g \wedge \forall\, (f \mathrel{\widehat{=}} g) \tag{29}$$

Proof: We show ($\Rightarrow$), the converse is similar.

$$\begin{array}{lll}
f = g & \Rightarrow\langle \text{Leibniz (12)} \rangle & \mathcal{D}\, f = \mathcal{D}\, g \wedge (x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, x = g\, x) \\
& \equiv\langle p \equiv (p \equiv 1) \rangle & \mathcal{D}\, f = \mathcal{D}\, g \wedge x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow (f\, x = g\, x) = 1 \\
& \equiv\langle \text{Def. (20, 14)} \rangle & \mathcal{D}\, f = \mathcal{D}\, g \wedge \\
& & x \in \mathcal{D}\, (f \mathrel{\widehat{=}} g) \Rightarrow (f \mathrel{\widehat{=}} g)\, x = (\mathcal{D}\, (f \mathrel{\widehat{=}} g)^\bullet 1)\, x \\
& \Rightarrow\langle \text{Extens. (13)} \rangle & \mathcal{D}\, f = \mathcal{D}\, g \wedge (f \mathrel{\widehat{=}} g) = \mathcal{D}\, (f \mathrel{\widehat{=}} g)^\bullet 1 \\
& \equiv\langle \text{Defin. } \forall \text{ (28)} \rangle & \mathcal{D}\, f = \mathcal{D}\, g \wedge \forall\, (f \mathrel{\widehat{=}} g)
\end{array}$$

Duality and other simple consequences of the axioms. By "head calculation", $\forall\, (X^\bullet 1) \equiv 1$ and $\exists\, (X^\bullet 0) \equiv 0$. Proof: (14), (28). In particular: $\forall\, \varepsilon \equiv 1$ and $\exists\, \varepsilon \equiv 0$ (proof: using $\varepsilon = \emptyset^\bullet 1 = \emptyset^\bullet 0$).

Illustrative of the algebraic style is the following theorem.

$$\text{Theorem, Duality (generalized De Morgan):}\quad \forall\, (\neg\, P) \;\equiv\; (\neg\, \exists)\, P \tag{30}$$

Proof:

$$\begin{array}{lll}
\forall\, (\neg\, P) & \equiv\langle \text{Def. } \forall \text{ (28)} \rangle & \neg\, P = \mathcal{D}\, (\neg\, P)^\bullet 1 \\
& \equiv\langle \text{Lemma A (31)} \rangle & \neg\, P = \mathcal{D}\, P^\bullet 1 \\
& \equiv\langle \text{Lemma B (31)} \rangle & P = \neg\, (\mathcal{D}\, P^\bullet 1) \\
& \equiv\langle \text{Lemma C (31), } 1 \in \mathcal{D}\, \neg \rangle & P = \mathcal{D}\, P^\bullet (\neg\, 1) \\
& \equiv\langle \neg\, 1 = 0, \text{ definit. } \exists \text{ (28)} \rangle & \neg\, (\exists\, P) \\
& \equiv\langle \text{Defin. } {}^- \text{ and } \exists\, P \in \mathcal{D}\, \neg \rangle & \neg\, \exists\, P
\end{array}$$

The lemmata are stated below, leaving the proofs as exercises.

$$\begin{array}{lll}
\text{Lemmata:} & \text{(A)}\;\; \mathcal{D}\, (\neg\, P) = \mathcal{D}\, P & \text{(B)}\;\; \neg\, P = Q \;\equiv\; P = \neg\, Q \\
& \text{(C)}\;\; x \in \mathcal{D}\, g \Rightarrow \overline{g}\, (X^\bullet x) = X^\bullet\, (g\, x) &
\end{array} \tag{31}$$

Given the preceding two representative proofs, further calculation rules will be stated without proof. Here are some initial distributivity rules.

$$\begin{array}{ll}
\textit{Collecting } \forall/\wedge: & \forall\, P \wedge \forall\, Q \Rightarrow \forall\,(P \mathbin{\widehat{\wedge}} Q) \\
\textit{Splitting } \forall/\wedge: & \mathcal{D}\, P = \mathcal{D}\, Q \Rightarrow \forall\,(P \mathbin{\widehat{\wedge}} Q) \Rightarrow \forall\, P \wedge \forall\, Q \\
\textit{Distributivity } \forall/\wedge: & \mathcal{D}\, P = \mathcal{D}\, Q \Rightarrow (\forall\,(P \mathbin{\widehat{\wedge}} Q) \equiv \forall\, P \wedge \forall\, Q) \\
\textit{Collecting } \exists/\vee: & \mathcal{D}\, P = \mathcal{D}\, Q \Rightarrow \exists\, P \vee \exists\, Q \Rightarrow \exists\,(P \mathbin{\widehat{\vee}} Q) \\
\textit{Splitting } \exists/\vee: & \exists\,(P \mathbin{\widehat{\vee}} Q) \Rightarrow \exists\, P \vee \exists\, Q \\
\textit{Distributivity } \exists/\vee: & \mathcal{D}\, P = \mathcal{D}\, Q \Rightarrow (\exists\,(P \mathbin{\widehat{\vee}} Q) \equiv \exists\, P \vee \exists\, Q)
\end{array}$$

Rules for equal predicates and isotony rules are the following.

$$\begin{array}{lll}
\textit{Equal predicates under } \forall: & \mathcal{D}\, P = \mathcal{D}\, Q \;\Rightarrow\; \forall\,(P \mathbin{\widehat{\equiv}} Q) \Rightarrow (\forall\, P \equiv \forall\, Q) \\
\textit{Equal predicates under } \exists: & \mathcal{D}\, P = \mathcal{D}\, Q \;\Rightarrow\; \forall\,(P \mathbin{\widehat{\equiv}} Q) \Rightarrow (\exists\, P \equiv \exists\, Q) \\
\textit{Isotony of } \forall/\Rightarrow: & \mathcal{D}\, Q \subseteq \mathcal{D}\, P \;\Rightarrow\; \forall\,(P \mathbin{\widehat{\Rightarrow}} Q) \Rightarrow (\forall\, P \Rightarrow \forall\, Q) \\
\textit{Isotony of } \exists/\Rightarrow: & \mathcal{D}\, P \subseteq \mathcal{D}\, Q \;\Rightarrow\; \forall\,(P \mathbin{\widehat{\Rightarrow}} Q) \Rightarrow (\exists\, P \Rightarrow \exists\, Q)
\end{array}$$

The latter two help chaining proof steps: $\forall\,(P \mathbin{\widehat{\Rightarrow}} Q)$ justifies $\forall\, P \Rightarrow \forall\, Q$ or $\exists\, P \Rightarrow \exists\, Q$ if the stated set inclusion for the domains holds.

The following theorem generalizes $\forall\,(X \bullet 1) \equiv 1$ and $\exists\,(X \bullet 0) \equiv 0$.

THEOREM, Constant Predicates:

$$\forall\,(X \bullet p) \equiv X = \emptyset \vee p \quad \text{and} \quad \exists\,(X \bullet p) \equiv X \neq \emptyset \wedge p \tag{32}$$

More distributivity laws. The main laws are the following.

$$\begin{array}{ll}
\textit{Distributivity } \vee/\forall: & \forall\,(q \mathbin{\vec{\vee}} P) \equiv q \vee \forall\, P \\
\textit{L(eft)-distrib. } \Rightarrow/\forall: & \forall\,(q \mathbin{\vec{\Rightarrow}} P) \equiv q \Rightarrow \forall\, P \\
\textit{R(ight)-distr. } \Rightarrow/\exists: & \forall\,(P \mathbin{\overleftarrow{\Rightarrow}} q) \equiv \exists\, P \Rightarrow q \\
\textit{P(seudo)-dist. } \wedge/\forall: & \forall\,(q \mathbin{\vec{\wedge}} P) \equiv (q \wedge \forall\, P) \vee \mathcal{D}\, P = \emptyset
\end{array}$$

We present the same laws in pointwise form, assuming v not free in q.

$$\begin{array}{ll}
\textit{Distributivity } \vee/\forall: & \forall\,(v : S \,.\, q \vee p) \equiv q \vee \forall\, v : S \,.\, p \\
\textit{L(eft)-distrib. } \Rightarrow/\forall: & \forall\,(v : S \,.\, q \Rightarrow p) \equiv q \Rightarrow \forall\, v : S \,.\, p \\
\textit{R(ight)-distr. } \Rightarrow/\exists: & \forall\,(v : S \,.\, p \Rightarrow q) \equiv \exists\,(v : S \,.\, p) \Rightarrow q \\
\textit{P(seudo)-dist. } \wedge/\forall: & \forall\,(v : S \,.\, q \wedge p) \equiv (q \wedge \forall\, v : S \,.\, p) \vee S = \emptyset
\end{array}$$

Here are the corresponding laws for $\exists$ (in point-free form only).

$$\begin{array}{ll}
\textit{Distributivity } \wedge/\exists: & \exists\,(q \mathbin{\vec{\wedge}} P) \equiv q \wedge \exists\, P \\
\textit{P(seudo)-dist. } \vee/\exists: & \exists\,(q \mathbin{\vec{\vee}} P) \equiv (q \vee \exists\, P) \wedge \mathcal{D}\, P \neq \emptyset \\
\textit{L(eft)-P-distrib. } \Rightarrow/\exists: & \exists\,(q \mathbin{\vec{\Rightarrow}} P) \equiv (q \Rightarrow \exists\, P) \wedge \mathcal{D}\, P \neq \emptyset \\
\textit{R(ight)-P-distr. } \Rightarrow/\forall: & \exists\,(P \mathbin{\overleftarrow{\Rightarrow}} q) \equiv (\forall\, P \Rightarrow q) \wedge \mathcal{D}\, P \neq \emptyset
\end{array}$$

Instantiation and generalization. The following theorem replaces axioms of traditional formal logic. It is proven from (28) using (12, 13).

$$\text{THEOREM,} \quad \text{Instantiation:} \quad \forall\, P \Rightarrow e \in \mathcal{D}\, P \Rightarrow P\, e \tag{33}$$

$$\text{Generalization:} \quad p \Rightarrow v \in \mathcal{D}\, P \;\Rightarrow P\, v \vdash p \Rightarrow \forall\, P \tag{34}$$

v being a fresh variable. Two typical proof techniques are captured by

$$\text{METATHEOREM, } \forall\text{-introduction/removal:} \quad \text{assuming fresh } v,$$
$$p \Rightarrow \forall\, P \text{ is a theorem iff } p \Rightarrow v \in \mathcal{D}\, P \Rightarrow P\, v \text{ is a theorem.} \tag{35}$$

$$\text{METATHEOREM, Witness:} \quad \text{assuming fresh } v,$$
$$\exists\, P \Rightarrow p \text{ is a theorem iff } v \in \mathcal{D}\, P \Rightarrow P\, v \Rightarrow p \text{ is a theorem.} \tag{36}$$

Significance: for $p = 1$, (35) reflects typical implicit use of generalization: to prove $\forall\, P$, prove $v \in \mathcal{D}\, P \Rightarrow P\, v$, or assume $v \in \mathcal{D}\, P$ and prove $P\, v$.

Also, (36) formalizes a well-known informal proof scheme: to prove $\exists\, P \Rightarrow p$, "take" a v in $\mathcal{D}\, P$ satisfying $P\, v$ (the "witness") and prove p.

As expected, we allow weaving (34) into a calculation chain in the following way, called *generalization of the consequent*: for fresh v,

$$\begin{array}{ll} p & \Rightarrow\langle \text{Calculation yielding } v \in \mathcal{D}\, P \Rightarrow P\, v \rangle \quad v \in \mathcal{D}\, P \Rightarrow P\, v \\ & \Rightarrow\langle \text{Generalization of the consequent} \rangle \quad \forall\, P. \end{array} \tag{37}$$

This convention (37) is used in the derivation of a few more basic calculation rules; it is rarely (if ever) appropriate beyond.

Trading. An example of using (37) is in the proof of the following.

$$\text{THEOREM, Trading under } \forall\text{:} \quad \forall\, P_Q \;\equiv\; \forall\, (Q \mathrel{\widehat{\Rightarrow}} P) \tag{38}$$

Proof: We prove only $\Rightarrow$, the converse $\Leftarrow$ being similar.

$$\begin{array}{lll} \forall\, P_Q & \Rightarrow\langle \text{Instantiation (33)} \rangle & x \in \mathcal{D}\, (P_Q) \Rightarrow P_Q\, x \\ & \equiv\langle \text{Definition } \downarrow \text{ (15)} \rangle & x \in \mathcal{D}\, P \cap \mathcal{D}\, Q \wedge Q\, x \Rightarrow P\, x \\ & \equiv\langle \text{Shunting } \wedge \text{ to } \Rightarrow \rangle & x \in \mathcal{D}\, P \cap \mathcal{D}\, Q \Rightarrow Q\, x \Rightarrow P\, x \\ & \equiv\langle \text{Axiom } \widehat{\ }\text{, remark} \rangle & x \in \mathcal{D}\, (Q \mathrel{\widehat{\Rightarrow}} P) \Rightarrow (Q \mathrel{\widehat{\Rightarrow}} P)\, x \\ & \Rightarrow\langle \text{Gen. conseq. (37)} \rangle & \forall\, (Q \mathrel{\widehat{\Rightarrow}} P) \end{array}$$

From (38) and using duality (30), one can prove the $\exists$-counterpart.

$$\text{THEOREM, Trading under } \exists\text{:} \quad \exists\, P_Q \;\equiv\; \exists\, (Q \mathrel{\widehat{\wedge}} P) \tag{39}$$

3.2 Expanding the toolkit of calculation rules

Building a full toolkit is beyond the scope of this tutorial and fits better in a textbook. Therefore, we just complement the preceding section with some guidelines and observations the reader will find sufficient for expanding the toolkit as needed.

Quantifiers applied to abstraction and tuples. With abstractions we synthesize or recover commonly used notations. For instance, letting $R := x : X \,.\, r$ and $P := x : X \,.\, p$ in the trading theorem (38) yields

$$\begin{array}{lrcl} \text{Trading (pointwise form):} & \forall\,(x : X \wedge r \,.\, p) & \equiv & \forall\,(x : X \,.\, r \Rightarrow p) \\ & \exists\,(x : X \wedge r \,.\, p) & \equiv & \exists\,(x : X \,.\, r \wedge p) \end{array} \tag{40}$$

For a tuple p, q of booleans, $\forall\,(p, q) \;\equiv\; p \wedge q$ and $\exists\,(p, q) \;\equiv\; p \vee q$.

A few more selected rules for $\forall$. We express them in both styles.

(i) Algebraic style. Legend: let P and Q be *predicates*, R a family of predicates (i.e., $R\,x$ is a predicate for any x in $\mathcal{D}\,R$), and S a relation. The *currying* operator $—^{\mathsf{C}}$ maps a function f with domain $X \times Y$ into a higher-order function f^{C} defined by $f^{\mathsf{C}} = x : X \,.\, y : Y \,.\, f\,(x, y)$. The *range* operator $\mathcal{R}$ is defined by $y \in \mathcal{R}\,f \;\equiv\; \exists\,x : \mathcal{D}\,f \,.\, f\,x = y$.

Merge rule	$P \mathbin{©} Q \Rightarrow \forall\,(P \cup Q) = \forall\,P \wedge \forall\,Q$
Transposition	$\forall\,(\forall \circ R) = \forall\,(\forall \circ R^{\mathsf{T}})$
Nesting	$\forall\,S = \forall\,(\forall \circ S^{\mathsf{C}})$
Composition rule	$\forall\,P \;\equiv\; \forall\,(P \circ f)$ provided $\mathcal{D}\,P \subseteq \mathcal{R}\,f$ (proof later)
One-point rule	$\forall\,P_{=e} \;\equiv\; e \in \mathcal{D}\,P \Rightarrow P\,e$

(ii) Using dummies. Legend: let p, q and r be $\mathbb{B}$-valued *expressions*, and assume the usual restrictions on types and free occurrences.

Domain split	$\forall\,(x : X \cup Y \,.\, p) \;\equiv\; \forall\,(x : X \,.\, p) \wedge \forall\,(x : Y \,.\, p)$
Dummy swap	$\forall\,(x : X \,.\, \forall\,y : Y \,.\, p) \;\equiv\; \forall\,(y : Y \,.\, \forall\,x : X \,.\, p)$
Nesting	$\forall\,((x, y) : X \times Y \,.\, p) \;\equiv\; \forall\,(x : X \,.\, \forall\,y : Y \,.\, p)$
Dummy change	$\forall\,(y : \mathcal{R}\,f \,.\, p) \;\equiv\; \forall\,(x : \mathcal{D}\,f \,.\, p[^{y}_{f\,x}])$
One-point rule	$\forall\,(x : X \wedge x = e \,.\, p) \;\equiv\; e \in X \Rightarrow p[^{x}_{e}]$

The one-point rule is found very important in applications. Being an equivalence, it is stronger than instantiation ($\forall\,P \Rightarrow e \in \mathcal{D}\,P \Rightarrow P\,e$).

A variant: the *half-pint rule*: $\forall\,(x : \mathcal{D}\,P \,.\, P\,x \Rightarrow x = e) \Rightarrow \exists\,P \Rightarrow P\,e$.

Swapping quantifiers and function comprehension. Dummy swap $\forall\,(x : X \,.\, \forall\,y : Y \,.\, p) \;\equiv\; \forall\,(y : Y \,.\, \forall\,x : X \,.\, p)$ and its dual for $\exists$ take care of "homogeneous" swapping. For mixed swapping in one direction,

$$\text{THEOREM, Swap } \forall \text{ out:} \quad \exists\,(y : Y \,.\, \forall\,x : X \,.\, p) \Rightarrow \forall\,(x : X \,.\, \exists\,y : Y \,.\, p) \tag{41}$$

The converse does not hold, but the following is a "pseudo-converse". Axiom, *Function comprehension*: for any relation $—R— : Y \times X \to \mathbb{B}$,

$$\forall\,(x : X \,.\, \exists\,y : Y \,.\, y\;R\;x) \Rightarrow \exists\,f : X \to Y \,.\, \forall\,x : X \,.\, (f\,x)\;R\;x. \tag{42}$$

This axiom (whose converse is easy to prove) is crucial for implicit function definitions.

4. Generic Applications

Most of applied mathematics and computing can be presented as applications of generic functionals and functional predicate calculus. This first batch of applications is generic and useful in any domain.

4.1 Applications to functions and functionals

Function range and applications. We define the range operator

$$\text{Function range } \mathcal{R}\text{:}\quad y \in \mathcal{R}\, f \;\equiv\; \exists\,(x : \mathcal{D}\, f \,.\, f\, x = e) \tag{43}$$

In point-free style: $e \in \mathcal{R}\, f \;\equiv\; \exists\,(f \overleftarrow{=} e)$. Now we can prove the

$$\begin{array}{ll}\textsc{Theorem}, \text{ Composition rule} & \text{(i) } \forall\, P \Rightarrow \forall\,(P \circ f) \\ & \text{(ii) } \mathcal{D}\, P \subseteq \mathcal{R}\, f \Rightarrow (\forall\,(P \circ f) \;\equiv\; \forall\, P)\end{array} \tag{44}$$

We prove the common part; items (i) and (ii) follow in 1 more step each.

$$\begin{array}{lll}\forall\,(P \circ f) & \equiv \langle \text{Definition } \circ \rangle & \forall\, x : \mathcal{D}\, f \wedge f\, x \in \mathcal{D}\, P \,.\, P\,(f\, x) \\ & \equiv \langle \text{Trading sub } \forall \rangle & \forall\, x : \mathcal{D}\, f \,.\, f\, x \in \mathcal{D}\, P \Rightarrow P\,(f\, x) \\ & \equiv \langle \text{One-point rule} \rangle & \forall\, x : \mathcal{D}\, f \,.\, \forall\, y : \mathcal{D}\, P \,.\, y = f\, x \Rightarrow P\, y \\ & \equiv \langle \text{Swap under } \forall \rangle & \forall\, y : \mathcal{D}\, P \,.\, \forall\, x : \mathcal{D}\, f \,.\, y = f\, x \Rightarrow P\, y \\ & \equiv \langle \text{R-dstr. } \Rightarrow/\exists \rangle & \forall\, y : \mathcal{D}\, P \,.\, \exists\,(x : \mathcal{D}\, f \,.\, y = f\, x) \Rightarrow P\, y \\ & \equiv \langle \text{Definition } \mathcal{R} \rangle & \forall\, y : \mathcal{D}\, P \,.\, y \in \mathcal{R}\, f \Rightarrow P\, y\end{array}$$

The dual is $\exists\,(P \circ f) \Rightarrow \exists\, P$ and $\mathcal{D}\, P \subseteq \mathcal{R}\, f \Rightarrow \exists\,(P \circ f) \;\equiv\; \exists\, P$.

An important application is expressing *set comprehension*. Introducing $\{—\}$ as an operator *fully interchangeable* with $\mathcal{R}$, expressions like $\{2, 3, 5\}$ and $\mathit{Even} = \{m : \mathbb{Z} \,.\, 2 \cdot m\}$ have a familiar form and meaning.

Indeed, since tuples are functions, $\{e, e', e''\}$ denotes a set by listing its elements. Also, $k \in \{m : \mathbb{Z} \,.\, 2 \cdot m\} \;\equiv\; \exists\, m : \mathbb{Z} \,.\, k = 2 \cdot m$ by (43). To cover common forms (without their flaws), abstraction has two variants:

$$\begin{array}{lll} e \mid x : X & \text{stands for} & x : X \,.\, e \\ x : X \mid p & \text{stands for} & x : X \wedge p \,.\, x, \end{array}$$

which synthesizes expressions like $\{2 \cdot m \mid m : \mathbb{Z}\}$ and $\{m : \mathbb{N} \mid m < n\}$.

Now binding is always trouble-free, even in $\{n : \mathbb{Z} \mid n \in \mathit{Even}\} = \{n : \mathit{Even} \mid n \in \mathbb{Z}\}$ and $\{n \in \mathbb{Z} \mid n : \mathit{Even}\} \neq \{n \in \mathit{Even} \mid n : \mathbb{Z}\}$.

All desired calculation rules follow from predicate calculus by the axiom for $\mathcal{R}$. A repetitive pattern is captured by the following property.

$$\textsc{Theorem}, \text{ Set comprehension:}\quad e \in \{x : X \mid p\} \;\equiv\; e \in X \wedge p[^x_e] \tag{45}$$

$$\begin{array}{lll}\text{Proof: } e \in \{x : X \wedge p \,.\, x\} & \equiv \langle \text{Def. range (43)} \rangle & \exists\, x : X \wedge p \,.\, x = e \\ & \equiv \langle \text{Trading, twice} \rangle & \exists\, x : X \wedge x = e \,.\, p \\ & \equiv \langle \text{One-point rule} \rangle & e \in X \wedge p[^x_e\end{array}$$

A generic function inverse (—$^-$). For *any* function f,

$$\mathcal{D}\, f^- = \mathsf{Bran}\, f \quad \text{and} \quad y \in \mathcal{D}\, f^- \Rightarrow f\,(f^-\, y) = y. \tag{46}$$

with, for Bdom (*bijectivity domain*) and Bran (*bijectivity range*),

$$\mathsf{Bdom}\, f \;=\; \{x : \mathcal{D}\, f \mid \forall\, x' : \mathcal{D}\, f \,.\, f\, x' = f\, x \Rightarrow x' = x\} \tag{47}$$

$$\mathsf{Bran}\, f \;=\; \{f\, x \mid x : \mathsf{Bdom}\, f\}. \tag{48}$$

Elastic extensions of generic functionals. *Elastic merge* ($\bigcup\!\!\!\cdot$) is defined in 2 parts to avoid clutter. For any function family f,

$$y \in \mathcal{D}\,(\bigcup\!\!\!\cdot\, f) \quad \equiv \tag{49}$$

$$y \in \bigcup(\mathcal{D} \circ f) \wedge \forall\,(x, x') : (\mathcal{D}\, f)^2 \,.\, y \in \mathcal{D}\,(f\, x) \cap \mathcal{D}\,(f\, x') \Rightarrow f\, x\, y = f\, x'\, y$$

$$y \in \mathcal{D}\,(\bigcup\!\!\!\cdot\, f) \quad \Rightarrow \quad \forall\, x : \mathcal{D}\, f \,.\, y \in \mathcal{D}\,(f\, x) \Rightarrow \bigcup\!\!\!\cdot\, f\, y = f\, x\, y \tag{50}$$

$\mathcal{D}\, f$ need not be discrete. Any function g satisfies $g = \bigcup\!\!\!\cdot\, x : \mathcal{D}\, g \,.\, x \mapsto g\, x$ and $g^- = \bigcup\!\!\!\cdot\, x : \mathcal{D}\, g \,.\, g\, x \mapsto x$; especially the latter is remarkable.

Elastic compatibility (©) For any function family f

$$\copyright\, f \quad \equiv \quad \forall\,(x, y) : (\mathcal{D}\, f)^2 \,.\, f\, x \copyright f\, y \tag{51}$$

In general, $\cup$ is not associative, but $\copyright\,(f, g, h) \Rightarrow (f \cup g) \cup h = f \cup (g \cup h)$.

A generic functional refining function types. The most common function typing operator is the *function arrow* ($\rightarrow$) defined by $f \in X \rightarrow Y \;\equiv\; \mathcal{D}\, f = X \wedge \mathcal{R}\, f \subseteq Y$, making $f\, x$ always of type Y. Similarly, $f \in X \nrightarrow Y \;\equiv\; \mathcal{D}\, f \subseteq X \wedge \mathcal{R}\, f \subseteq Y$ defines the *partial arrow*.

More refined is the *tolerance* concept [11]: given a family T of sets, called the *tolerance function*, then a function f *meets* tolerance T iff $\mathcal{D}\, f = \mathcal{D}\, T$ and $x \in \mathcal{D}\, f \cap \mathcal{D}\, T \Rightarrow f\, x \in T\, x$. We define an operator $\bigtimes$:

$$f \in \bigtimes T \;\equiv\; \mathcal{D}\, f = \mathcal{D}\, T \wedge \forall\, x : \mathcal{D}\, f \cap \mathcal{D}\, T \,.\, f\, x \in T\, x \tag{52}$$

Equivalently, $\bigtimes T = \{f : \mathcal{D}\, T \rightarrow \bigcup\, T \mid \forall\,(f \mathrel{\widehat{\in}} T)\}$. The tolerance can be "exact": $f = g \;\equiv\; f \in \bigtimes(\iota \circ g)$ (exercise).

Since $\bigtimes(A, B) = A \times B$ (exercise), we call $\bigtimes$ the *generalized functional Cartesian product*. Another property is $A \rightarrow B = \bigtimes(A^\bullet\, B)$.

Clearly, $\bigtimes(a : A \,.\, B_a) \;=\; \{f : A \rightarrow \bigcup\, a : A \,.\, B_a \;\mid\; \forall\, a : A \,.\, f\, a \;\in\; B_a\}$. This point-wise form is a *dependent type* [19] or *product of sets* [30]. We write $A \ni a \rightarrow B_a$ as a shorthand for $\bigtimes a : A \,.\, B_a$, especially in chained dependencies: $A \ni a \rightarrow B_a \ni b \rightarrow C_{a,b} \;=\; \bigtimes a : A \,.\, \bigtimes b : B_a \,.\, C_{a,b}$. This is (intentionally) similar to, but *not* the same as, the function arrow.

Remarkable is the following simple explicit formula for the inverse: $\bigtimes^- S = x : \bigcap(f : S \,.\, \mathcal{D}\, f) \,.\, \{f\, x \mid f : S\}$ for any S in $\mathcal{D} \bigtimes^-$ (exercise).

4.2 Calculating with relations

Concepts. Given set X, we let $\mathsf{pred}_X := X \to \mathbb{B}$ and $\mathsf{rel}_X := X^2 \to \mathbb{B}$. We list some potential characteristics of relations R in rel_X, formalizing each property by a predicate $P : \mathsf{rel}_X \to \mathbb{B}$ and an expression for $P\,R$. Point-free forms as in [1] are left as an exercise.

Characteristic	P	Expression for $P\,R$ (pointwise)
Reflexivity	Refl	$\forall\, x : X \,.\, x\,R\,x$
Transitivity	Trns	$\forall\,(x,y,z) : X^3 \,.\, x\,R\,y \Rightarrow y\,R\,z \Rightarrow x\,R\,z$
Symmetry	Symm	$\forall\,(x,y) : X^2 \,.\, x\,R\,y \Rightarrow y\,R\,x$
Antisymmetry	Ants	$\forall\,(x,y) : X^2 \,.\, x\,R\,y \Rightarrow y\,R\,x \Rightarrow x = y$
Equivalence	EQ	$\mathsf{Refl}\,R \wedge \mathsf{Trns}\,R \wedge \mathsf{Symm}\,R$
Partial order	PO	$\mathsf{Refl}\,R \wedge \mathsf{Trns}\,R \wedge \mathsf{Ants}\,R$
Well-founded	WF	$\forall\, S : \mathcal{P}\,X \,.\, S \neq \emptyset \Rightarrow \exists\, x : X \,.\, x\ ismin_R\ S$

In the last line, $x\ ismin_\prec\ S \equiv x \in S \wedge \forall\, y : X \,.\, y \prec x \Rightarrow y \notin S$. We often write $\prec$ for R. Here *ismin* had type $\mathsf{rel}_X \to X \times \mathcal{P}\,X \to \mathbb{B}$, but predicate transformers of type $\mathsf{rel}_X \to \mathsf{pred}_X \to \mathsf{pred}_X$ are more elegant. Hence we use the latter in the following characterizations of extremal elements.

Characteristic	Symbol	Definition
Lower bound	lb	$\mathsf{lb}_\prec\,P\,x \equiv \forall\, y : X \,.\, P\,y \Rightarrow x \prec y$
Least	lst	$\mathsf{lst}_\prec\,P\,x \equiv P\,x \wedge \mathsf{lb}\,P\,x$
Minimal	min	$\mathsf{min}_\prec\,P\,x \equiv P\,x \wedge \forall\, y : X \,.\, y \prec x \Rightarrow \neg\,(P\,y)$
Upper bound	ub	$\mathsf{ub}_\prec\,P\,x \equiv \forall\, y : X \,.\, P\,y \Rightarrow y \prec x$
Greatest	gst	$\mathsf{gst}_\prec\,P\,x \equiv P\,x \wedge \mathsf{ub}\,P\,x$
Maximal	max	$\mathsf{max}_\prec\,P\,x \equiv P\,x \wedge \forall\, y : X \,.\, x \prec y \Rightarrow \neg\,(P\,y)$
Least u.b.	lub	$\mathsf{lub}_\prec = \mathsf{lst}_\prec \circ \mathsf{ub}_\prec$
Greatest l.b.	glb	$\mathsf{glb}_\prec = \mathsf{gst}_\prec \circ \mathsf{lb}_\prec$

Calculational reasoning about extremal elements. In this example, we derive some properties used later. A predicate $P : \mathsf{pred}_X$ is *isotonic* for a relation $—\prec— : \mathsf{rel}_X$ iff $\forall\,(x,y) : X^2 \,.\, x \prec y \Rightarrow P\,x \Rightarrow P\,y$.

$$\text{THEOREM: For any } —\prec— : \mathsf{rel}_X \quad \text{and} \quad P : \mathsf{pred}_X, \tag{53}$$

0. If $\prec$ is reflexive, then $\forall\,(y : X \,.\, x \prec y \Rightarrow P\,y) \Rightarrow P\,x$.
1. If $\prec$ is transitive, then $\mathsf{ub}_\prec\,P$ is isotonic w.r.t. $\prec$.
2. If P is isotonic for $\prec$, then $\mathsf{lst}_\prec\,P\,x \equiv P\,x \wedge \forall\,(y : X \,.\, P\,y \equiv x \prec y)$.
3. If $\mathsf{Refl} \prec$ and $\mathsf{Trns} \prec$, then $\mathsf{lub}_\prec\,P\,x \equiv \forall\,(y : X \,.\, \mathsf{ub}\,P\,y \equiv x \prec y)$.
4. If $\prec$ is antisymmetric, then $\mathsf{lst}_\prec\,P\,x \wedge \mathsf{lst}_\prec\,P\,y \Rightarrow x = y$.

Replacing lb by ub and so on yields complementary theorems.

Proofs. For part 0, instantiate with $y := x$. For part 1, we assume $\prec$ transitive and prove $x \prec y \Rightarrow \mathsf{ub}_\prec\, P\, x \Rightarrow \mathsf{ub}_\prec\, P\, y$ in shunted form.

$$\begin{array}{lll}
\mathsf{ub}_\prec\, P\, x & \Rightarrow\langle p \Rightarrow q \Rightarrow p\rangle & x \prec y \Rightarrow \mathsf{ub}_\prec\, P\, x \\
& \equiv\langle \text{Definition } \mathsf{ub}\rangle & x \prec y \Rightarrow \forall\, z : X \,.\, P\, z \Rightarrow z \prec x \wedge 1 \\
& \equiv\langle p \Rightarrow e[^v_1 = e[^v_p\rangle & x \prec y \Rightarrow \forall\, z : X \,.\, P\, z \Rightarrow z \prec x \wedge x \prec y \\
& \Rightarrow\langle \text{Transitiv. } \prec\rangle & x \prec y \Rightarrow \forall\, z : X \,.\, P\, z \Rightarrow z \prec y \\
& \equiv\langle \text{Definition } \mathsf{ub}\rangle & x \prec y \Rightarrow \mathsf{ub}_\prec\, P\, y
\end{array}$$

For part 2, we assume P isotonic and calculate $\mathsf{lst}_\prec\, P\, x$

$$\begin{array}{lll}
\mathsf{lst}_\prec\, P\, x & \equiv\langle \text{Defin. } \mathsf{lst},\, \mathsf{lb}\rangle & P\, x \wedge \forall\, y : X \,.\, P\, y \Rightarrow x \prec y \\
& \equiv\langle \text{Modus Pon.}\rangle & P\, x \wedge (P\, x \Rightarrow \forall\, y : X \,.\, P\, y \Rightarrow x \prec y) \\
& \equiv\langle \text{L-dstr. } \Rightarrow/\forall\rangle & P\, x \wedge \forall\, y : X \,.\, P\, x \Rightarrow P\, y \Rightarrow x \prec y \\
& \equiv\langle \text{Isotony of } P\rangle & P\, x \wedge \forall\, y : X \,.\, (P\, x \Rightarrow P\, y \Rightarrow x \prec y) \\
& & \qquad\qquad \wedge\, (P\, x \Rightarrow x \prec y \Rightarrow P\, y) \\
& \equiv\langle \text{L-dstr. } \Rightarrow/\wedge\rangle & P\, x \wedge \forall\, y : X \,.\, P\, x \Rightarrow (P\, y \Rightarrow x \prec y) \\
& & \qquad\qquad \wedge\, (x \prec y \Rightarrow P\, y) \\
& \equiv\langle \text{Mut. implic.}\rangle & P\, x \wedge \forall\, y : X \,.\, P\, x \Rightarrow (P\, y \equiv x \prec y) \\
& \equiv\langle \text{L-dstr. } \Rightarrow/\forall\rangle & P\, x \wedge (P\, x \Rightarrow \forall\, y : X \,.\, P\, y \equiv x \prec y) \\
& \equiv\langle \text{Modus Pon.}\rangle & P\, x \wedge \forall\, (y : X \,.\, P\, y \equiv x \prec y)
\end{array}$$

Part 3 combines 0, 1, 2. Part 4 (uniqueness) is a simple exercise and justifies defining the usual glb (and lub) functionals $\sqcap$ (and $\sqcup$).

4.3 Induction principles

A relation $\prec : X^2 \to \mathbb{B}$ is said to *support induction* iff $\mathsf{SI}\ (\prec)$, where

$$\mathsf{SI}\ (\prec) \quad \equiv \quad \forall\, P : \mathsf{pred}_X \,.\, \forall\, (x : X \,.\, \forall\, (y : X_{\prec x} \,.\, P\, y) \Rightarrow P\, x) \Rightarrow \forall\, P$$

One can show $\mathsf{SI}\ (\prec) \equiv \mathsf{WF}\ (\prec)$; a calculational proof is given in [10].

Examples are the familiar strong and weak induction over $\mathbb{N}$. One of the axioms for natural numbers is: every nonempty subset of $\mathbb{N}$ has a *least* element under $\leq$ or, equivalently, a *minimal* element under $<$. Strong induction over $\mathbb{N}$ is obtained by taking $<$ for $\prec$, yielding

$$\forall\, P : \mathsf{pred}_{\mathbb{N}} \,.\, \forall\, (n : \mathbb{N} \,.\, \forall\, (m : \mathbb{N} \,.\, m < n \Rightarrow P\, m) \Rightarrow P\, n) \Rightarrow \forall\, P \qquad (54)$$

Weak induction over $\mathbb{N}$ can be obtained from (54) or as follows. Define $\prec$ by $m \prec n \equiv m + 1 = n$ and prove that $\mathsf{WF}\,(\prec)$. Hence, from (53),

$$\forall\, P : \mathsf{pred}_{\mathbb{N}} \,.\, P\, 0 \wedge \forall\, (n : \mathbb{N} \,.\, P\, n \Rightarrow P\, (n + 1)) \Rightarrow \forall\, P. \qquad (55)$$

Another example is structural induction over data structures (see later).

An important preparatory step to avoid errors in proofs by induction is always making the induction predicate P and all quantification explicit, and avoiding vague designations such as "induction over n". This is especially important in case other variables are involved.

5. Applications in Computing

5.1 Calculating with data structures

Unifying principle: data types as function spaces. Tuples, sequences and so on are ubiquitous in computing as well as mathematics, and derive most benefit from being defined as functions. This allows sharing the collection of generic functionals and their calculation rules.

Sequences. This term encompasses tuples, arrays, lists and so on. A *sequence* is function with domain $\square\, n$ for some $n : \mathbb{N} \cup \iota\, \infty$. We define (i) the *block* operator $\square : \mathbb{N} \cup \iota\, \infty \to \mathcal{P}\, \mathbb{N}$ with $\square\, n = \{m : \mathbb{N} \mid m < n\}$, as in $\square\, 2 = \mathbb{B}$, (ii) the *power* of a set by $A \uparrow n = \square\, n \to A = \bigtimes (\square\, n \bullet A)$, also written A^n, (iii) the *length* operator $(\#)$ by $\#\, x = n \equiv \mathcal{D}\, x = \square\, n$. This also covers *arrays* in programming.

The set of *lists* over A, written A^*, is defined by $A^* = \bigcup\, n : \mathbb{N} \,.\, A^n$. Infinite lists are covered by $A^\infty = \mathbb{N} \to A$. *Tuples* are similarly defined as functions. *Tuple types* are then types of the form $\bigtimes S$ where S is any sequence of nonempty sets. Clearly $\bigtimes S \subseteq (\bigcup\, S)^{\#\, S} \subseteq (\bigcup\, S)^*$.

As in [7], we define the list operators *prefixing* $(- \succ\!\!- -)$ and *concatenation* $(- +\!\!+ -)$ for any a and any sequences x and y by

$$a \succ\!\!- x \quad = \quad i : \square\, (\#\, x + 1) \,.\, (i = 0)\, ?\, a \nmid x\, (i - 1) \tag{56}$$

$$x +\!\!+ y \quad = \quad i : \square\, (\#\, x + \#\, y) \,.\, (i < \#\, x)\, ?\, x\, i \nmid y\, (i - \#\, x) \tag{57}$$

The formulas $\varepsilon +\!\!+ y = y$ and $(a \succ\!\!- x) +\!\!+ y = a \succ\!\!- (x +\!\!+ y)$ can be seen as either theorems derived from (57) or a recursive definition replacing (57). The (weak) structural induction principle for finite lists over A is

$$\forall\, P : \mathsf{pred}_{A^*} \,.\, P\, \varepsilon \wedge \forall\, (x : A^* \,.\, P\, x \Rightarrow \forall\, a : A \,.\, P\, (a \succ\!\!- x)) \Rightarrow \forall\, P \tag{58}$$

The notation x, y, z is complemented by $\tau\, a = 0 \mapsto a$ covering length 1.

Records and other structures. *Records* as in PASCAL [21] are expressed via the funcart product $\bigtimes$ as *functions* whose domain is a set of field labels constituting an *enumeration type*. For instance, letting `name` and `age` be elements of an enumeration type,

$$\mathit{Person} := \bigtimes (\mathtt{name} \mapsto \mathbb{A}^* \cup \mathtt{age} \mapsto \mathbb{N})$$

defines a function type such that an identifier $\mathit{person} : \mathit{Person}$ satisfies $\mathit{person}\, \mathtt{name} \in \mathbb{A}^*$ and $\mathit{person}\, \mathtt{age} \in \mathbb{N}$. Obviuosly, by defining $\mathsf{record}\, F = \bigtimes (\bigcup\!\!\!\cdot\, F)$, one can also write $\mathit{Person} := \mathsf{record}\, (\mathtt{name} \mapsto \mathbb{A}^*, \mathtt{age} \mapsto \mathbb{N})$.

Trees are functions whose domains are *branching structures*, i.e., sets of sequences describing the path from the root to a leaf in the obvious way (for any branch labeling). Other structures are covered similarly

Example: relational databases. The record type declaration **def** $CID := \mathsf{record}\,(\mathsf{code} \mapsto Code, \mathsf{name} \mapsto \mathbb{A}^*, \mathsf{inst} \mapsto Staff, \mathsf{prrq} \mapsto Code^*)$ specifies the type of tables of the form

Code	Name	Instructor	Prerequisites
CS100	Basic Mathematics for CS	R. Barns	none
MA115	Introduction to Probability	K. Jason	MA100
CS300	Formal Methods in Engineering	R. Barns	CS100, EE150
...	...	...	

All typical query-operators are subsumed by generic functionals:

The *selection*-operator (σ) is subsumed by $\sigma\,(S, P) = S \downarrow P$.

The *projection*-operator (π) is subsumed by $\pi\,(S, F) = \{r \rceil F \mid r : S\}$.

The *join*-operator ($\bowtie$) is subsumed by $S \bowtie T = S \otimes T$.

Here $\otimes$ is the generic *function type merge* operator, defined as in [11] by $S \otimes T = \{s \cup t \mid (s, t) : S \times T \mathbin{\&} s \mathbin{©} t\}$. Its elastic extension (exercise) is the generic variant of Van den Beuken's function type merge [31]. Note that function type merge is associative, although function merge is not.

5.2 Systems specification and implementation

Abstract specification. An abstract specification should be free of implementation decisions. We consider *sorting* as an example.

Let A be a set with total order $\sqsubseteq$. Sorting means that the result is ordered and has the same contents. We formalize this by two functions: $ndesc : A^* \to \mathbb{B}$ ("nondescending") and $\S : A^* \to A \to \mathbb{N}$ ("inventory") such that $\S\, x\, a$ is the number of times a is present in x

$$\begin{array}{rcl} ndesc\, x & \equiv & \forall\,(i, j) : (\mathcal{D}\, x)^2 \,.\, i \le j \Rightarrow x\, i \sqsubseteq x\, j \\ \S\, x\, a & = & \sum i : \mathcal{D}\, x \,.\, a = f\, i \end{array}$$

Our general definition of $\sum$ has 3 parts: $\sum \varepsilon = 0$ and $\sum (a \mapsto c) = c$ and $\sum (f \uplus g) = \sum f + \sum g$ for any a, any numeric c and any number-valued functions f and g with finite nonintersecting domains. We specify

$$\textbf{spec}\ sort : A^* \to A^*\ \textbf{with}\ ndesc\,(sort\, x) \mathbin{\&} \S\,(sort\, x) = \S\, x$$

The general form **spec** $v : X \mathbin{\&} p$ introduces v with axiom $v \in X \mathbin{\&} p$.

Implementation. A typical (functional) program implementation is

$$\begin{array}{l} \textbf{def}\ qsort : A^* \to A^*\ \textbf{with} \\ \quad qsort\, \varepsilon = \varepsilon \mathbin{\&} \\ \quad qsort\,(a \succ\!\!- x) = qsort\, u +\!\!+ \tau\, a +\!\!+ qsort\, v\ \textbf{where}\ u, v := split\, a\, x \\ \textbf{def}\ split : A \to A^* \to A^* \times A^*\ \textbf{with} \\ \quad split\, a\, \varepsilon = \varepsilon, \varepsilon \mathbin{\&} \\ \quad split\, a\,(b \succ\!\!- x) = (a \sqsubseteq b)\,?\,(u, b \succ\!\!- v) \mid (b \succ\!\!- u, v) \\ \qquad\qquad\qquad \textbf{where}\ u, v := split\, a\, x \end{array}$$

Verification. We must prove $asc\,(qsort\,x)$ and $\S\,(qsort\,x) = \S\,x$.

Here we give an outline only; more details are found in [8]. Based on problem analysis, we introduce functions *le* and *ge*, both of type $A \times A^* \to \mathbb{B}$, with $a\,le\,x \;\equiv\; \forall\, i : \mathcal{D}\,x\,.\,a \sqsubseteq x\,i$ and $a\,ge\,x \;\equiv\; \forall\, i : \mathcal{D}\,x\,.\,x\,i \sqsubseteq a$.

Properties most relevant here are expressed by two lemmata: for any x and y in A^* and a in A, and letting $u, v := split\,a\,x$, we can show:

	Split lemma	Concatenation lemma
(a)	$\S\,u \mathbin{\widehat{+}} \S\,v = \S\,x$	$\S\,(x +\!\!+ y) = \S\,x \mathbin{\widehat{+}} \S\,y$
(b)	$a\,ge\,u \wedge a\,le\,v$	$asc\,(x +\!\!+ \tau\,a +\!\!+ y) \;\equiv\; asc\,x \wedge asc\,y \wedge a\,ge\,x \wedge a\,le\,y$

Note that $\S\,(x +\!\!+ y) = \S\,x \mathbin{\widehat{+}} \S\,y$ together with $\S\,\varepsilon = A^\bullet\,0$ and $\S\,(\tau\,a) = (=a)\rceil A$ makes $\S$ into a *list homomorphism* as defined in [3].

The properties $asc\,\varepsilon \;\equiv\; 1$ and $asc\,(a \mathbin{\succ\!\!-} x) \;\equiv\; a\,le\,x \wedge asc\,x$ and the mixed property $\S\,x = \S\,y \Rightarrow ge\,x = ge\,y \wedge le\,x = le\,y$ are the ingredients for making the proof of $asc\,(qsort\,x)$ and $\S\,(qsort\,x) = \S\,x$ simple exercise.

A hardware-flavored example. Here we consider a data flow example whose implementation style is typical for hardware but also for dataflow languages such as LabVIEW [4]. Let the specification be

$$\textbf{spec}\; f : \mathbb{N} \to A \;\textbf{with}\; f\,n = (n = 0)\,?\,a \mathbin{\dagger} g\,(f\,(n-1)), \tag{59}$$

for a given set A, element a in A and function $g : A \to A$. By calculation,

$$\begin{array}{rll} f\,n & =\langle \text{Def. } f\rangle & (n=0)\,?\,a \mathbin{\dagger} g\,(f\,(n-1)) \\ & =\langle \text{Def. } \circ\rangle & (n=0)\,?\,a \mathbin{\dagger} (g \circ f)\,(n-1) \\ & =\langle \text{Def. D}\rangle & \mathrm{D}_a\,(g \circ f)\,n \\ & =\langle \text{Def. } {}^{-}\rangle & \mathrm{D}_a\,(\overline{g}\;f)\,n \\ & =\langle \text{Def. } \circ\rangle & (\mathrm{D}_a \circ \overline{g}\,)\,f\,n, \end{array}$$

yielding the *fixpoint equation* $f = (\mathrm{D}_a \circ \overline{g}\,)\,f$ by extensionality. The function $\mathrm{D} : A \to A^* \to A^+$ is defined by $\mathrm{D}_a\,x\,n = (n=0)\,?\,a \mathbin{\dagger} x\,(n-1)$.

Let the variable n be associated with discrete time, then D is the *unit delay* element. The block diagram in Fig. 1 realizes $f = (\mathrm{D}_a \circ \overline{g}\,)\,f$.

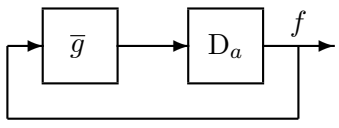

Figure 1. Signal flow realization of specification (59)

5.3 Formal semantics and programming theories

Abstract syntax. This example shows how generic functionals subsume existing ad hoc conventions as in [24]. For *aggregate constructs* and *list productions*, we use the $\times$-operator as embodied in the record and list types. For *choice productions* where a disjoint union is needed, we define a generic operator $\big|$ such that, for any family F of types,

$$\big|F = \bigcup x : \mathcal{D}\, F \,.\, \{x \mapsto y \mid y : F\, x\} \tag{60}$$

simply by analogy with $\bigcup F = \bigcup (x : \mathcal{D}\, F \,.\, F\, x) = \bigcup x : \mathcal{D}\, F \,.\, \{y \mid y : F\, x\}$. Typical examples are (with field labels from an enumeration type):

def *Program* := record $(\mathit{declarations} \mapsto \mathit{Dlist}, \mathit{body} \mapsto \mathit{Instruction})$
def *Dlist* := D^*
def D := record $(v \mapsto \mathit{Variable}, t \mapsto \mathit{Type})$
def *Instruction* := $\mathit{Skip} \cup \mathit{Assignment} \cup \mathit{Compound} \cup$ etc.

For disjoint union one can write $\mathit{Skip} \mid \mathit{Assignment} \mid \mathit{Compound}$ etc.

Instances of programs, declarations, etc. can be defined as

def p : *Program* **with** $p = \mathit{declarations} \mapsto dl \cup \mathit{body} \mapsto \mathit{instr}$

Static semantics. Subsuming [24], the *validity* of declaration lists (no double declarations) and the variable inventory are expressed by

def $\mathit{Vdcl} : \mathit{Dlist} \to \mathbb{B}$ **with** $\mathit{Vdcl}\ dl = \mathsf{inj}\,(dl^{\mathsf{T}} v)$
def $\mathit{Var} : \mathit{Dlist} \to \mathcal{P}\ \mathit{Variable}$ **with** $\mathit{Var}\ dl = \mathcal{R}\,(dl^{\mathsf{T}} v)$

The *type map* (from variables to types) [24] of a valid declaration list is

def $\mathit{typmap} : \mathit{Dlist}_{Vdcl} \ni dl \to \mathit{Var}\ dl \to \mathit{Tval}$ **with**
$\mathit{typmap}\ dl = \mathit{tval} \circ (dl^{\mathsf{T}}\, t) \circ (dl^{\mathsf{T}}\, v)^{-}$

The function *merge* obviates case expressions. For instance, assume

def *Expression* := $\mathit{Constant} \cup \mathit{Variable} \cup \mathit{Applic}$
def *Constant* := $\mathit{IntCons} \cup \mathit{BoolCons}$
def *Applic* := record $(\mathit{op} \mapsto \mathit{Operator}, \mathit{term} \mapsto \mathit{Expression}, \mathit{term}' \mapsto \mathit{Expression})$

Then, letting $\mathit{Tmap} := \bigcup dl : \mathit{Dlist}_{Vdcl} \,.\, \mathit{typmap}\ dl$ and $\mathit{Tval} := \{it, bt, ut\}$ (integer, boolean, undefined), the type of expressions is defined by

def $\mathit{Texp} : \mathit{Tmap} \to \mathit{Expression} \to \mathit{Tval}$ **with**
$\mathit{Texp}\ tm = (c : \mathit{IntCons} \,.\, it) \cup (c : \mathit{BoolCons} \,.\, bt)$
$\cup\ (v : \mathit{Variable} \,.\, ut) \lhd\!\!\!\cdot\ tm$
$\cup\ (a : \mathit{Applic} \,.\, (a\ \mathit{op} \in \mathit{Arith\_op})\ ?\ it \nmid bt)$

jointly with an "expression validity" function, left as an exercise [11].

Deriving programming theories. Functional predicate calculus subsumes special program logics by deriving their axioms as theorems.

Let the *state* s be the tuple made of the program variables (and perhaps auxiliary ones), and $\mathbf{S}$ its type. Variable reuse is made unambiguous by priming: $\grave{s}$ denotes the state before and s' the state after the execution of a command. We use $s \bullet e$ as shorthand for $s : \mathbf{S} \,.\, e$.

If C is the set of commands, $\mathsf{R} : C \to \mathbf{S}^2 \to \mathbb{B}$ and $\mathsf{T} : C \to \mathbf{S} \to \mathbb{B}$ are defined such that the effect of a command c can be described by two equations: $\mathsf{R}\, c\, (\grave{s}, s')$ for state change and $\mathsf{T}\, c\, \grave{s}$ for termination. For technical reasons, we sometimes we write $\mathsf{R}\, c\, (s, s')$ and $\mathsf{T}\, c\, s$ by α-conversion.

Here is an example for Dijkstra's *guarded command* language [13].

Syntax	Behavior (program equations or equivalent program)	
Command c	State change $\mathsf{R}\, c\, (s, s')$	Termination $\mathsf{T}\, c\, s$
$v := e$	$s' = s[^v_e$	1
`skip`	$s' = s$	1
`abort`	0	0
$c' \,;\, c''$	$\exists\, t \bullet \mathsf{R}\, c'\, (s, t) \wedge \mathsf{R}\, c''\, (t, s')$	$\mathsf{T}\, c'\, s \wedge \forall\, t \bullet \mathsf{R}\, c'\, (s, t) \Rightarrow \mathsf{T}\, c''\, t$
`if` $[]\, i : I \,.\, b_i$ `->` c'_i `fi`	$\exists\, i : I \,.\, b_i \wedge \mathsf{R}\, c'_i\, (s, s')$	$\exists\, b \wedge \forall\, i : I \,.\, b_i \Rightarrow \mathsf{T}\, c'_i\, s$
`do` b `->` c' `od`	`if` $\neg\, b$ `->` `skip` $[]$ b `->` $(c' \,;\, c)$ `fi`	

Let the state before and after executing c satisfy a (*antecondition*) and p (*postcondition*) respectively, then Hoare-semantics is captured by

$$
\begin{array}{rll}
\{a\}\ c\ \{p\} & \equiv \forall\, \grave{s} \bullet \forall\, s' \bullet a[^s_{\grave{s}} \wedge \mathsf{R}\, c\, (\grave{s}, s') \ \Rightarrow\ p[^s_{s'} & \text{“partial correctness”} \\
\mathit{Term}\, c\, a & \equiv \forall\, \grave{s} \bullet a[^s_{\grave{s}} \Rightarrow \mathsf{T}\, c\, \grave{s} & \text{“termination”} \\
[a]\ c\ [p] & \equiv \{a\}\ c\ \{p\} \wedge \mathit{Term}\, c\, a & \text{“total correctness”}
\end{array}
$$

Now everything is reduced to functional predicate calculus. Calculating

$$
\begin{array}{rll}
[a]\ c\ [p] & \equiv \langle \text{Definition} \rangle & \forall\, \grave{s} \bullet \forall\, s' \bullet a[^s_{\grave{s}} \Rightarrow (\mathsf{R}\, c\, (\grave{s}, s') \ \Rightarrow\ p[^s_{s'}) \wedge \mathsf{T}\, c\, \grave{s} \\
& \equiv \langle \text{Ldistr. } \Rightarrow/\forall \rangle & \forall\, \grave{s} \bullet a[^s_{\grave{s}} \Rightarrow \forall\, s' \bullet (\mathsf{R}\, c\, (\grave{s}, s') \ \Rightarrow\ p[^s_{s'}) \wedge \mathsf{T}\, c\, \grave{s} \\
& \equiv \langle \text{Pdistr. } \wedge/\forall \rangle & \forall\, \grave{s} \bullet a[^s_{\grave{s}} \Rightarrow \forall\, (s' \bullet \mathsf{R}\, c\, (\grave{s}, s') \ \Rightarrow\ p[^s_{s'}) \wedge \mathsf{T}\, c\, \grave{s} \\
& \equiv \langle \text{Change var} \rangle & \forall\, s \bullet a \Rightarrow \forall\, (s' \bullet \mathsf{R}\, c\, (s, s') \ \Rightarrow\ p[^s_{s'}) \wedge \mathsf{T}\, c\, s
\end{array}
$$

and theorem (53) justifies capturing Dijkstra-style semantics [13] by

$$
\begin{array}{rll}
\mathsf{wla}\, c\, p & \equiv \forall\, s' \bullet \mathsf{R}\, c\, (s, s') \ \Rightarrow\ p[^s_{s'} & \text{“weakest liberal antecondition”} \\
\mathsf{wa}\, c\, p & \equiv \mathsf{wla}\, c\, p \wedge \mathsf{T}\, c\, s & \text{“weakest antecondition”}
\end{array}
$$

From this, we obtain by calculation in functional predicate calculus [12]

$$
\begin{array}{rcl}
\mathsf{wa}\, [\![v := e]\!]\, p & \equiv & p[^v_e \\
\mathsf{wa}\, [\![c' \,;\, c'']\!]\, p & \equiv & \mathsf{wa}\, c'\, (\mathsf{wa}\, c''\, p) \\
\mathsf{wa}\, [\![\texttt{if}\ []\, i : I \,.\, b_i\ \texttt{->}\ c'_i\ \texttt{fi}]\!]\, p & \equiv & \exists\, b \wedge \forall\, i : I \,.\, b_i \Rightarrow \mathsf{wa}\, c'_i\, p \\
\mathsf{wa}\, [\![\texttt{do}\ b\ \texttt{->}\ c'\ \texttt{od}]\!]\, p & \equiv & \exists\, n : \mathbb{N} \,.\, w^n\, (\neg\, b \wedge p) \quad \text{defining } w \text{ by} \\
w\, q & \equiv & (\neg\, b \wedge p) \vee (b \wedge \mathsf{wa}\, c'\, q) \quad .
\end{array}
$$

6. Applications in continuous mathematics

6.1 An example in mathematical analysis

The topic is *adjacency* [22], here expressed by a predicate transformer since predicates were found to yield more elegant formulations than sets.

$$\textbf{def}\ \mathsf{ad} : \mathsf{pred}_{\mathbb{R}} \to \mathsf{pred}_{\mathbb{R}}\ \textbf{with}\ \mathsf{ad}\, P\, v \ \equiv\ \forall\, \epsilon : \mathbb{R}_{>0} \,.\, \exists\, x : \mathbb{R}_P \,.\, |x - v| < \epsilon$$

The concepts "open" and "closed" are similarly defined by predicates.

$$\begin{array}{l}\textbf{def}\ \mathsf{open} : \mathsf{pred}_{\mathbb{R}} \to \mathbb{B}\ \textbf{with}\\ \quad \mathsf{open}\, P \ \equiv\ \forall\, v : \mathbb{R}_P \,.\, \exists\, \epsilon : \mathbb{R}_{>0} \,.\, \forall\, x : \mathbb{R} \,.\, |x - v| < \epsilon \Rightarrow P\, x\\ \textbf{def}\ \mathsf{closed} : \mathsf{pred}_{\mathbb{R}} \to \mathbb{B}\ \textbf{with}\ \mathsf{closed}\, P \ \equiv\ \mathsf{open}\, (\neg P)\end{array}$$

An exercise in [22] is proving the *closure property* $\mathsf{closed}\, P \ \equiv\ \mathsf{ad}\, P = P$. The calculation, assuming the (easily proven) lemma $P\, v \Rightarrow \mathsf{ad}\, P\, v$, is

$$\begin{array}{ll}&\mathsf{closed}\, P\\ \equiv\langle \text{Def. } \mathsf{closed}\rangle & \mathsf{open}\, (\neg P)\\ \equiv\langle \text{Def. } \mathsf{open}\rangle & \forall\, v : \mathbb{R}_{\neg P} \,.\, \exists\, \epsilon : \mathbb{R}_{>0} \,.\, \forall\, x : \mathbb{R} \,.\, |x - v| < \epsilon \Rightarrow \neg P\, x\\ \equiv\langle \text{Trading } \forall\rangle & \forall\, v : \mathbb{R} \,.\, \neg P\, v \Rightarrow \exists\, \epsilon : \mathbb{R}_{>0} \,.\, \forall\, x : \mathbb{R} \,.\, |x - v| < \epsilon \Rightarrow \neg P\, x\\ \equiv\langle \text{Contrap.}\rangle & \forall\, v : \mathbb{R} \,.\, \neg \exists\, (\epsilon : \mathbb{R}_{>0} \,.\, \forall\, x : \mathbb{R} \,.\, P\, x \Rightarrow \neg\, (|x - v| < \epsilon)) \Rightarrow P\, v\\ \equiv\langle \text{Duality}\rangle & \forall\, v : \mathbb{R} \,.\, \forall\, (\epsilon : \mathbb{R}_{>0} \,.\, \exists\, x : \mathbb{R} \,.\, P\, x \wedge |x - v| < \epsilon) \Rightarrow P\, v\\ \equiv\langle \text{Def. } \mathsf{ad}\rangle & \forall\, v : \mathbb{R} \,.\, \mathsf{ad}\, P\, v \Rightarrow P\, v\\ \equiv\langle \text{Lemma}\rangle & \forall\, v : \mathbb{R} \,.\, \mathsf{ad}\, P\, v \ \equiv\ P\, v \quad .\end{array}$$

6.2 An example about transform methods

This example formalizes Laplace transforms via Fourier transforms. In doing so, we pay attention to using functionals in a formally correct way. In particular, we avoid common abuses of notation like $\mathcal{F}\,\{f(t)\}$ and write $\mathcal{F}\, f\, \omega$ instead. As a consequence, in the definitions

$$\mathcal{F}\, f\, \omega = \int_{-\infty}^{+\infty} e^{-j\cdot\omega\cdot t} \cdot f\, t \cdot \mathrm{d}\, t \qquad \mathcal{F}'\, g\, t = \frac{1}{2\cdot\pi} \cdot \int_{-\infty}^{+\infty} e^{j\cdot\omega\cdot t} \cdot g\, \omega \cdot \mathrm{d}\, \omega$$

the bindings are clear and unambiguous without contextual information. This is important in formal calculation. For what follows, we assume some familiarity with transforms via the usual informal treatments.

Given $\mathrm{l}\_ : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$ with $\mathrm{l}_\sigma\, t \ =\ (t \ <\ 0)\,?\,0 \ \mathbin{\mathrm{!}}\ e^{-\sigma\cdot t}$ (conditioning functions), we define the Laplace-transform $\mathcal{L}$ of a given function f by:

$$\mathcal{L}\, f\, (\sigma + j\cdot\omega) \ \ =\ \ \mathcal{F}\ (\mathrm{l}_\sigma \mathbin{\hat{\cdot}} f)\, \omega \tag{61}$$

for real σ and ω, with suitable conditions on σ to make $\mathrm{l}_\sigma \mathbin{\hat{\cdot}} f$ Fourier transformable. With $s := \sigma + j\cdot\omega$ we obtain $\mathcal{L}\, f\, s = \int_0^{+\infty} f\, t \cdot e^{-s\cdot t} \cdot \mathrm{d}\, t$.

The converse $\mathcal{L}'$ is specified by $\mathcal{L}'\,(\mathcal{L}\,f)\,t = f\,t$ for all $t \geq 0$, weakened where $\mathrm{l}_\sigma \mathbin{\hat{\cdot}} f$ is discontinuous: in these points information is lost by $\mathcal{F}$, and $\mathcal{F}' \circ \mathcal{F}$ reproduces a given function exactly in the continuous parts only. For these (nonnegative) t-values,

$$\begin{array}{lcll}
\mathcal{L}'\,(\mathcal{L}\,f)\,t & = & \langle\text{Specification}\rangle & f\,t \\
& = & \langle e^{\sigma\cdot t}\cdot \mathrm{l}_\sigma\,t = 1\rangle & e^{\sigma\cdot t}\cdot \mathrm{l}_\sigma\,t\cdot f\,t \\
& = & \langle\text{Definition } \mathbin{\hat{\cdot}}\rangle & e^{\sigma\cdot t}\cdot (\mathrm{l}_\sigma \mathbin{\hat{\cdot}} f)\,t \\
& = & \langle\text{Weakening}\rangle & e^{\sigma\cdot t}\cdot \mathcal{F}'\,(\mathcal{F}\,(\mathrm{l}_\sigma \mathbin{\hat{\cdot}} f))\,t \\
& = & \langle\text{Definition } \mathcal{F}'\rangle & e^{\sigma\cdot t}\cdot \frac{1}{2\cdot\pi}\cdot \int_{-\infty}^{+\infty} \mathcal{F}\,(\mathrm{l}_\sigma \mathbin{\hat{\cdot}} f)\,\omega\cdot e^{j\cdot\omega\cdot t}\cdot \mathrm{d}\,\omega \\
& = & \langle\text{Definition } \mathcal{L}\rangle & e^{\sigma\cdot t}\cdot \frac{1}{2\cdot\pi}\cdot \int_{-\infty}^{+\infty} \mathcal{L}\,f\,(\sigma + j\cdot\omega)\cdot e^{j\cdot\omega\cdot t}\cdot \mathrm{d}\,\omega \\
& = & \langle\text{Factor } e^{\sigma\cdot t}\rangle & \frac{1}{2\cdot\pi}\cdot \int_{-\infty}^{+\infty} \mathcal{L}\,f\,(\sigma + j\cdot\omega)\cdot e^{(\sigma+j\cdot\omega)\cdot t}\cdot \mathrm{d}\,\omega \\
& = & \langle s := \sigma + j\cdot\omega\rangle & \frac{1}{2\cdot\pi\cdot j}\cdot \int_{\sigma-j\cdot\infty}^{\sigma+j\cdot\infty} \mathcal{L}\,f\,s\cdot e^{s\cdot t}\cdot \mathrm{d}\,s
\end{array}$$

For $t < 0$, of course $\mathcal{L}'\,(\mathcal{L}\,f)\,t = 0$. The calculation shows how to derive the inverse transform using functionals in a formally correct way.

7. Some final notes on the Funmath formalism

The formalism used in this tutorial is called *Funmath*, a contraction of *Fun*ctional *math*ematics. It is not "yet another computer language", but an approach for designing formalisms by characterizing mathematical objects as functions whenever this is possible and useful. The latter is the case quite more often than common conventions suggest.

As we have seen, the language needs four constructs only:

0 *Identifier*: a constant or a variable, declared by a binding.

1 *Application*: a function with argument(s), as in $f\,x$ and $x \star y$.

2 *Tupling*, of the form e, e', e'', was briefly introduced in section 1.4.

3 *Abstraction*, of the form $v : X \mathbin{\wedge} p\,.\,e$, was introduced in section 2.1.

The calculation rules and their application were the main topic of this tutorial. Only function application requires a few additional notes.

Identifiers denoting functions are called *operators* The standard affix convention is *prefix*, as in $f\,x$. Other affix conventions can be specified by dashes in the binding introducing the operator, e.g., $\text{—}\star\text{—}$ for infix. Parentheses restore the standard prefix convention, e.g., $(\star)(x, y) = x \star y$.

Partial application is the following convention for omitting arguments. Let $\text{—}\star\text{—} : X \times Y \to Z$. For any $x : X$ and $y : Y$ we have $(x\star) \in Y \to Z$ with $(x\star)\,y = x \star y$ and $(\star y) \in X \to Z$ with $(\star y)\,x = x \star y$.

Argument/operator alternations of the form $x \star y \star z$ are called *variadic application* and (in Funmath) are *always* defined via an elastic extension: $x \star y \star z = F\,(x, y, z)$. An example is $x \wedge y \wedge z = \forall\,(x, y, z)$. This is not restricted to associative or commutative operators. For instance, letting con and inj be the *constant* and *injective* predicates over functions, we define $x = y = z \;\equiv\; \mathsf{con}\,(x, y, z)$ and $x \neq y \neq z \;\equiv\; \mathsf{inj}\,(x, y, z)$. The latter gives $x \neq y \neq z$ the most useful meaning (x, y, z distinct).

From the material in this tutorial, it is clear the language and the calculation rules jointly constitute a very broad-spectrum formalism.

References

[1] Chritiene Aarts, Roland Backhouse, Paul Hoogendijk, Ed Voermans and Jaap van der Woude, *A relational theory of data types*. Lecture notes, Eindhoven University (December 1992)

[2] Henk P. Barendregt, *The Lambda Calculus — Its Syntax and Semantics*, North-Holland (1984)

[3] Richard Bird, *Introduction to Functional Programming using Haskell*. Prentice Hall International Series in Computer Science, London (1998)

[4] Robert H. Bishop, *LabVIEW Student Edition*, Prentice Hall, N.J. (2001)

[5] Eerke Boiten and Bernhard Möller, *Sixth International Conference on Mathematics of Program Construction* (Conference announcement), Dagstuhl (2002). `http://www.cs.kent.ac.uk/events/conf/2002/mpc2002`

[6] Raymond T. Boute, "A heretical view on type embedding", *ACM Sigplan Notices 25*, pp. 22–28 (Jan. 1990)

[7] Raymond T. Boute, "Declarative Languages — still a long way to go", in: Dominique Borrione and Ronald Waxman, eds., *Computer Hardware Description Languages and their Applications*, pp. 185–212, North-Holland (1991)

[8] Raymond T. Boute, *Funmath illustrated: A Declarative Formalism and Application Examples*. Declarative Systems Series No. 1, Computing Science Institute, University of Nijmegen (July 1993)

[9] Raymond T. Boute, "Supertotal Function Definition in Mathematics and Software Engineering", *IEEE Transactions on Software Engineering, Vol. 26*, No. 7, pp. 662–672 (July 2000)

[10] Raymond Boute, *Functional Mathematics: a Unifying Declarative and Calculational Approach to Systems, Circuits and Programs — Part I: Basic Mathematics*. Course text, Ghent University (2002)

[11] Raymond T. Boute, "Concrete Generic Functionals: Principles, Design and Applications", in: Jeremy Gibbons and Johan Jeuring, eds., *Generic Programming*, pp. 89–119, Kluwer (2003)

[12] Raymond T. Boute, "Calculational semantics: deriving programming theories from equations by functional predicate calculus", Technical note B2004/02, INTEC, Universiteit Gent (2004) (submitted for publication to *ACM TOPLAS*)

[13] Edsger W. Dijkstra and Carel S. Scholten, *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin (1990)

[14] Edsger W. Dijkstra, *Under the spell of Leibniz's dream*. EWD1298 (April 2000). http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1298.pdf

[15] Ganesh Gopalakrishnan, *Computation Engineering: Formal Specification and Verification Methods* (Aug. 2003). http://www.cs.utah.edu/classes/cs6110/lectures/CH1/ch1.pdf

[16] David Gries, "Improving the curriculum through the teaching of calculation and discrimination", *Communications of the ACM 34*, 3, pp. 45–55 (March 1991)

[17] David Gries and Fred B. Schneider, *A Logical Approach to Discrete Math*, Springer-Verlag, Berlin (1993)

[18] David Gries, "The need for education in useful formal logic", *IEEE Computer 29*, 4, pp. 29–30 (April 1996)

[19] Keith Hanna, Neil Daeche and Gareth Howells, "Implementation of the Veritas design logic", in: Victoria Stavridou and Tom F. Melham and Raymond T. Boute, eds., *Theorem Provers in Circuit Design*, pp. 77–84. North Holland (1992)

[20] Eric C. R. Hehner, *From Boolean Algebra to Unified Algebra*. Internal Report, University of Toronto (June 1997, revised 2003)

[21] Kathleen Jensen and Niklaus Wirth, *PASCAL User Manual and Report*. Springer-Verlag, Berlin (1978)

[22] Serge Lang, *Undergraduate Analysis*. Springer-Verlag, Berlin (1983)

[23] Leslie Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education Inc. (2002)

[24] Bertrand Meyer, *Introduction to the Theory of Programming Languages*. Prentice Hall, New York (1991)

[25] David L. Parnas, "Education for Computing Professionals", *IEEE Computer 23*, 1, pp. 17–20 (January 1990)

[26] David L. Parnas, "Predicate Logic for Software Engineering", *IEEE Trans. SWE 19*, 9, pp. 856–862 (Sept. 1993)

[27] Raymond Ravaglia, Theodore Alper, Marianna Rozenfeld, Patrick Suppes, "Successful pedagogical applications of symbolic computation", in: N. Kajler, *Computer-Human Interaction in Symbolic Computation*. Springer, 1999. http://www-epgy.stanford.edu/research/chapter4.pdf

[28] J. Mike Spivey, *The Z notation: A Reference Manual*. Prentice-Hall (1989).

[29] Paul Taylor, *Practical Foundations of Mathematics* (second printing), No. 59 in *Cambridge Studies in Advanced Mathematics*, Cambridge University Press (2000); quotation from comment on chapter 1 in http://www.dcs.qmul.ac.uk/~pt/Practical_Foundations/html/s10.html

[30] Robert D. Tennent, *Semantics of Programming Languages*. Prentice-Hall (1991).

[31] Frank van den Beuken, *A Functional Approach to Syntax and Typing*, PhD thesis. School of Mathematics and Informatics, University of Nijmegen (1997)

[32] Jeannette M. Wing, "Weaving Formal Methods into the Undergraduate Curriculum", *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST)* pp. 2–7 (May 2000) http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/calder/www/amast00.html